Metrology of land plots
Rapport de projet

Arnaud Aujon, Matthieu Foucault, Rachid Hafiane, Milan Kabac
Encadré par : M. Pascal Desbarats
Sujet proposé par : M. Hugo Gimbert

5 avril 2011

Table des matières

Table des figures

Chapitre 1

Introduction

La métrologie est la science des mesures et ses applications. L'étude métrologique des structures agraires, permet de retracer l'histoire d'une ville ou d'une région en identifiant, grâce aux unités de mesure utilisées pour tracer les champs, quel civilisation a occupé une région sur une période donnée.

L'objectif de ce projet est de construire un outil informatique pour la métrologie sur plans géographiques en archéologie permettant de déterminer quel unité de mesure aurait pu être utilisé pour tracer les champs à partir des coordonnées GPS de ceux-ci et ensuite déduire la civilisation occupante de la région lors de la construction de ces champs.

Pour cela nous comparerons les dimensions des champs étudiées dans une région avec les unités de mesure utilisées à l'époque. La difficulté est que nous ne connaissons pas toutes les mesures utilisées, notre travail consistera alors à essayer de trouver une nouvelle unité de mesure commune aux champs et ayant pu être utilisée.

Ce problème de reconnaissance a déjà été traité, mais les moyens d'acquérir les données étaient différents. Par exemple, en 2006, Bescoby publie un article intitulé "Detecting Roman land boundaries in aerial photographs using Radon transforms"[**?**]. Dans cet article, Bescoby propose un moyen d'acquisition des données et surtout un moyen d'interprétation des résultats obtenus.

Chapitre 2

Cahier des charges

2.1 Étude des besoins

2.1.1 Définitions

1. **Entité** Une entité représente un élément géographique, soit un point GPS, une ligne ou une zone.
2. **Point GPS :** Un point GPS est identifié par deux coordonnées : la latitude, comprise entre -90°et 90°, et la longitude, comprise entre -180°et 180°.
3. **Ligne :** Une ligne est composée de deux points GPS (les extrémités).
4. **Zone :** Une zone est un regroupement de lignes et/ou d'autres zones.

2.1.2 Besoins non fonctionnels

Précision des distances calculées

Les distances calculées par le logiciel doivent être précises par rapport aux distances réelles, l'erreur doit être inférieure à 1 mètre pour une distance réel de 20 mètres, soit 5%.
Test de validation : Comparer les distances obtenues grâce au logiciel avec celles obtenues sur le terrain.

Ergonomie

Le logiciel doit être simple à utiliser et l'utilisateur (archéologue) doit se familiariser avec le logiciel rapidement.
Le logiciel doit être accessible aux utilisateurs francophones, anglophones et hispanophones.
Test de validation : Demander à des utilisateurs archéologues (étudiants et/ou professeurs) d'effectuer un travail (cf Tests) sur notre logiciel. L'utilisateur ne connaîtra pas le logiciel à l'avance et devra apprendre à l'utiliser (grâce à la documentation disponible) en une demi-heure. Les réponses des utilisateurs au questionnaire (cf Annexes) ainsi que leur commentaires seront les critères de validation de ce test.

Portabilité

Le logiciel doit être accessible par l'intermédiaire des navigateurs Mozilla-Firefox et Google-Chrome, et doit fonctionner sur les systèmes d'exploitation Linux, Microsoft : Windows XP, Windows 7 et sur Mac OS X.
Test de validation : Un scenario (cf Tests) sera réalisé avec les navigateurs et systèmes d'exploitation décrits ci-dessus.

2.1.3 Besoins fonctionnels

Identifier les unités de mesure utilisées

Le logiciel doit identifier les unités de mesures les plus probablement utilisées pour tracer les lignes d'une zone.
Niveau de difficulté : Moyen
Priorité : Élevée.
Test de validation : Comparer les résultats obtenus par notre logiciel avec ceux de sites déjà connus.

Déterminer une nouvelle unité de mesure

Il sera possible de calculer la longueur d'une nouvelle unité de mesure probablement plus utilisée que les unités de mesure déjà connues.
Niveau de difficulté : Moyen.
Priorité : Élevée.

Intégrer l'application dans un navigateur

L'interface devra être intégrée dans un navigateur.
Niveau de difficulté : Faible.
Priorité : Élevée.

Intégrer GoogleMaps

L'utilisateur doit pouvoir visualiser les entités géographiques créées, sur une carte GoogleMaps, et ajouter celles-ci directement sur la carte.
Niveau de difficulté Moyen.
Priorité : Élevée.
Limitation : Nécessite une connexion à Internet.
Risques et parades : Un risque important est présent sur l'utilisation de GoogleMaps, il n'est pas garanti qu'il existe une API pour intégrer GoogleMaps à un site web, ou que la licence soit compatible à celle de notre logiciel. Dans ce cas il faudrait utiliser un autre système de cartographie en ligne, comme par exemple Mappy.

Sauvegarder et charger un projet au format XML

Le logiciel doit pouvoir charger et sauvegarder des fichiers de projet au format XML, tout en respectant la norme XML 1.0 définie par le W3C (http ://www.w3.org/2001/03/webdata/xsv).
Niveau de difficulté : Faible.
Priorité : Moyenne.
Test de validation : Le projet, une fois chargé, doit être dans le même état qu'à l'instant où il a été sauvegardé.

Exporter et importer les entités au format KML

Le logiciel doit pouvoir exporter et importer des fichiers KML (format utilisé pour décrire des informations géographiques dans GoogleMaps) en respectant la norme KML 2.2 définie par le OGC (Open Geospatial Consortium).
Niveau de difficulté : Élevée.
Priorité : Moyenne.
Test de validation : Les fichiers générés doivent être lisibles par GoogleEarth, GoogleMaps ainsi qu'ArcView.

Éditer la liste d'unités de mesure

L'utilisateur pourra éditer la liste d'unités de mesure. Ces unités de mesure pourront être définies par rapport aux unités existantes.
Niveau de difficulté : Faible.
Priorité : Moyenne.
Test de validation : Il est possible de créer une liste d'au moins 100 unités de mesure différentes.

Générer un rapport

Le logiciel devra permettre la génération d'un rapport en PDF contenant les résultats obtenus ainsi qu'une visualisation de la carte.
Niveau de difficulté : Moyen.
Priorité : Faible.

Avoir un manuel en ligne

Le logiciel devra fournir un accès rapide à une documentation. Un tutoriel en ligne devra également expliquer pas à pas les différentes étapes de l'utilisation du logiciel.
Niveau de difficulté : Faible.
Priorité : Faible.

2.2 Cas d'utilisation

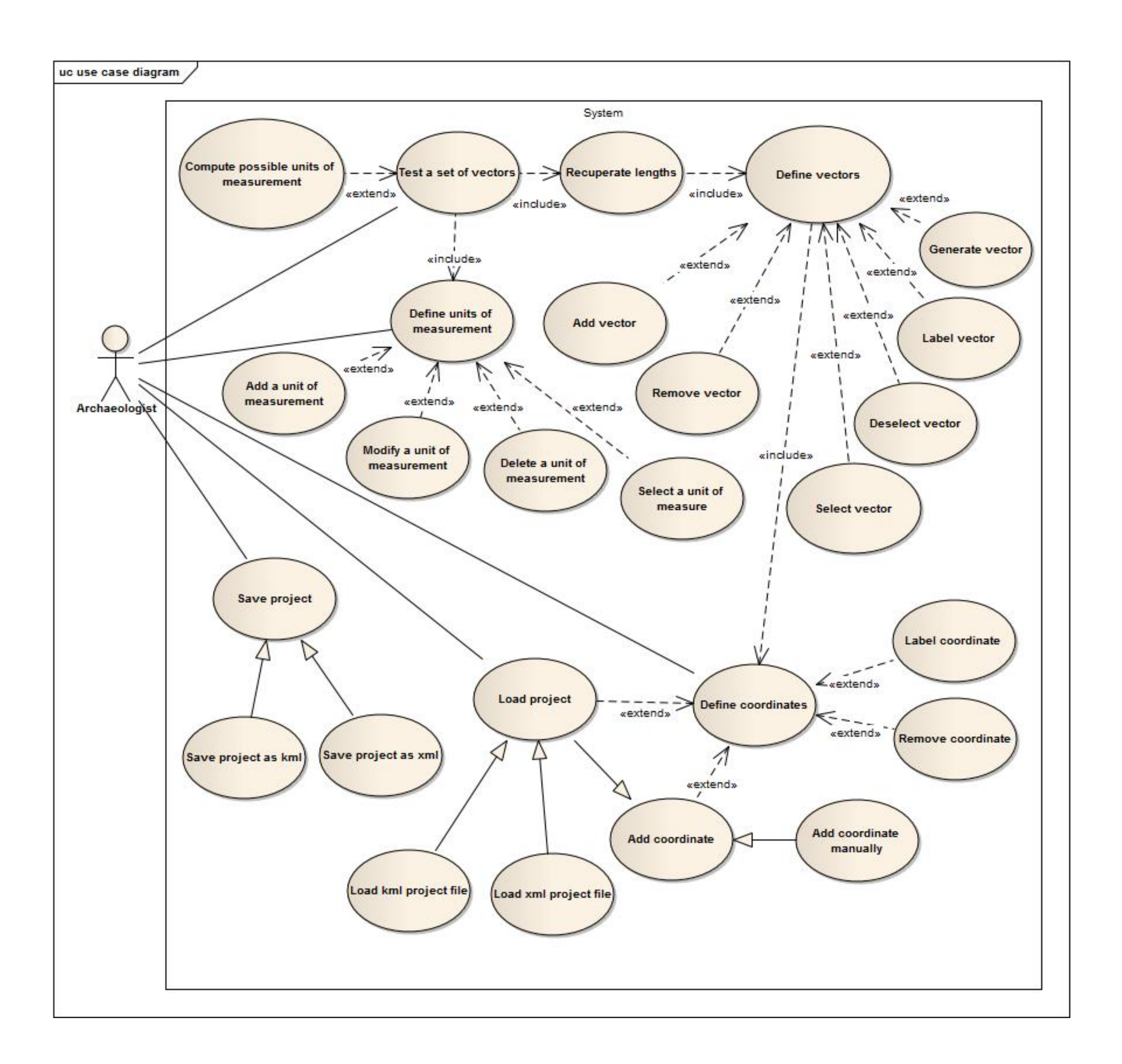


FIG. 2.1 – Diagramme de cas d'utilisation

Relation "include" : Les cas d'utilisation peuvent contenir les fonctionnalités d'un autre cas d'utilisation. Quand un cas d'utilisation est traité, tous les cas d'utilisation qui sont inclus doivent être exécutées afin de traiter le cas d'utilisation prévu.

Dans le diagramme de cas d'utilisation (figure 2.1) la relation "include" est utilisé par exemple quand un archéologue veut effectuer un test d'un ensemble de vecteurs. Pour exécuter le test, l'archéologue doit définir les coordonnées, les vecteurs et les mesures. Par la suite, l'application doit récupérer les longueurs correspondantes. Toutes ces fonctionnalités doivent être réalisées afin d'exécuter le cas d'utilisation demandé.

Relation "extend" : Un cas d'utilisation peut être utilisés pour étendre le comportement d'un autre cas d'utilisation. En d'autres termes, relation "extend" donne un comportement supplémentaire au cas d'utilisation parent. Nous pouvons prendre la définition de coordonnées par exemple. En définissant les coordonnées l'archéologue a la possibilité d'ajouter, de supprimer ou d'étiqueter une coordonnée.

Relation "generalization" : Généralisation entre des cas d'utilisation signifie en d'autres termes que le cas d'utilisation enfant hérite des propriétés et du comportement du cas d'utilisation parent et peut modifier le comportement du parent. Un cas d'utilisation de base peut être un cas d'utilisation abstrait ou concret. Si le cas d'utilisation de base est abstrait, il est nécessaire de définir au moins un cas d'utilisation spécialisé. On peut prendre le cas d'utilisation "Save project" comme example. Ce cas d'utilisation est abstrait, en d'autres terme si l'archéologue veut enregistrer le projet il doit choisir un des cas d'utilisation concret.

2.3 Prototype d'interface utilisateur

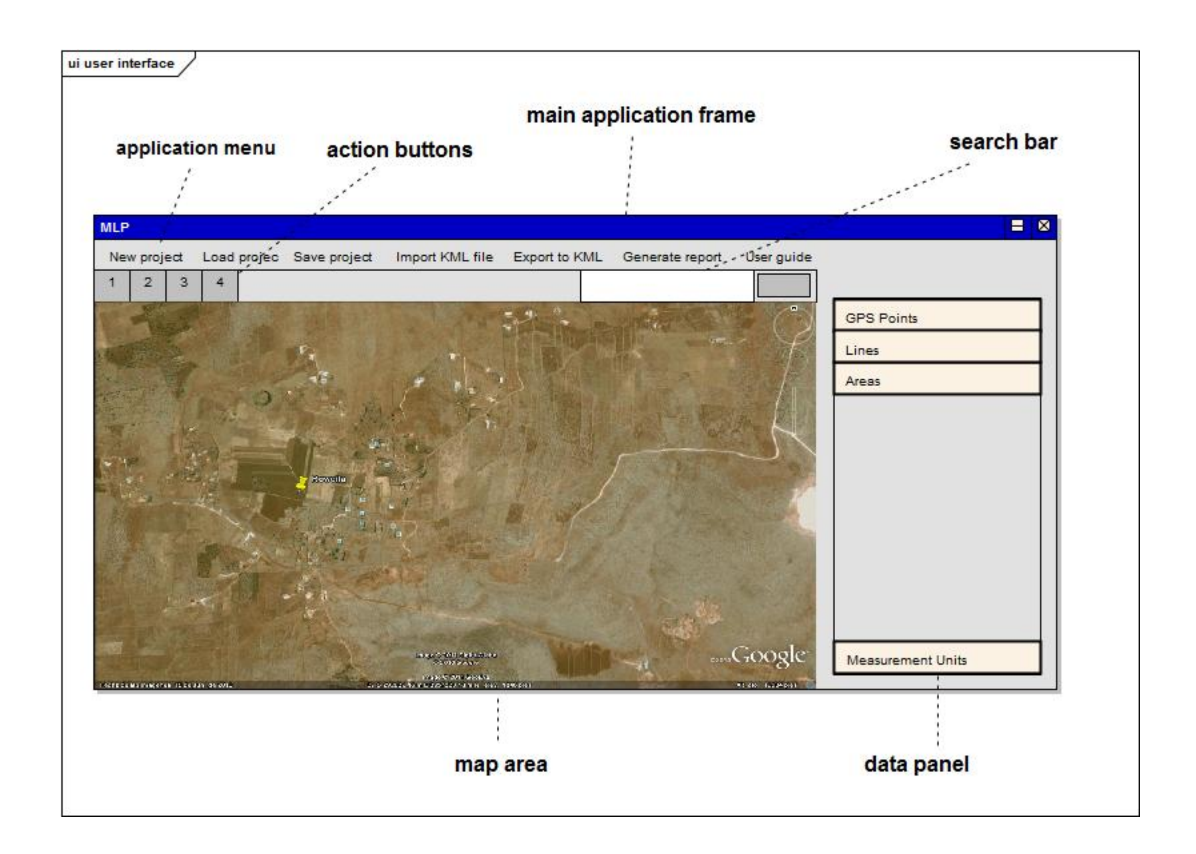


FIG. 2.2 – Prototype de l'interface utilisateur

L'interface utilisateur se compose de trois parties : le menu, la carte, et le panneau d'informations.

Les boutons du menu permettent respectivement de créer un nouveau projet, de charger un projet existant, de sauvegarder le projet en cours, d'importer et d'exporter les fichiers KML, de générer un rapport et d'afficher le manuel en ligne.

La carte, qui est contrôlable avec la souris, de la même manière qu'une carte GoogleMaps classique (molette de la souris pour zoomer, cliquer-glisser pour se déplacer), a également une barre d'outils associée, avec des boutons qui permettent d'ajouter et supprimer des points et des lignes, et un champ de recherche, qui permet à l'utilisateur de centrer la carte sur un lieu dont il a entré le nom.

La partie droite de l'interface graphique est composé d'un panneau contenant les données de l'application, (les entités géographiques et les unités de mesures), et permettant d'éditer ces données.

2.4 Planning prévisionnel

Le diagramme de Gantt suivant montre les nombres de développeurs associés à chaque tâche et la succession de celles-ci ainsi que leurs durées.

La semaine 1 correspond à la semaine du 31 janvier au 6 février.

A la fin de la semaine 5, une version alpha du logiciel, accompagnée d'une ébauche du manuel d'utilisateur sera présentée au client, afin de valider l'ergonomie de l'interface.

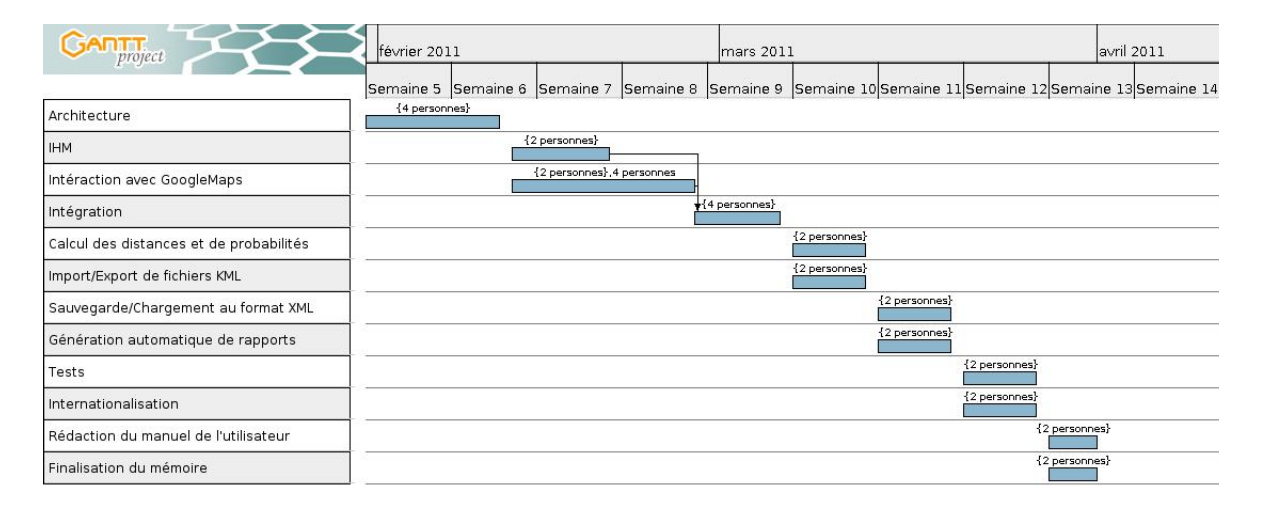


FIG. 2.3 – Diagramme de Gantt du planning prévisionnel

Chapitre 3

Exemples d'utilisation

3.1 Ajout, Suppression et Édition

3.1.1 A partir de la carte

Grâce aux boutons on peut ajouter ou supprimer des points ou des lignes dans la carte (voir fig. 3.1).

En cliquant sur une entité dans la carte une info-bulle s'affiche (voir : fig. 3.2).

3.1.2 Via les menus

À l'aide des panneaux d'informations (voir fig. 3.3) on peut voir les entités existantes, en supprimer, ajouter ou éditer.

Le panneau affichant les zones permet de réaliser l'analyse qui trouve les unités de mesures utilisées, dans la ou les zone(s) sélectionnée(s).

L'édition et l'ajout d'une entité se fait en éditant ses informations et son nom.

3.1.3 Les calculs

Les résultats s'affichent temporairement sur une PopUp (voir fig. 3.5).

3.1.4 La génération du rapport

La génération du rapport se fait en trois étape :

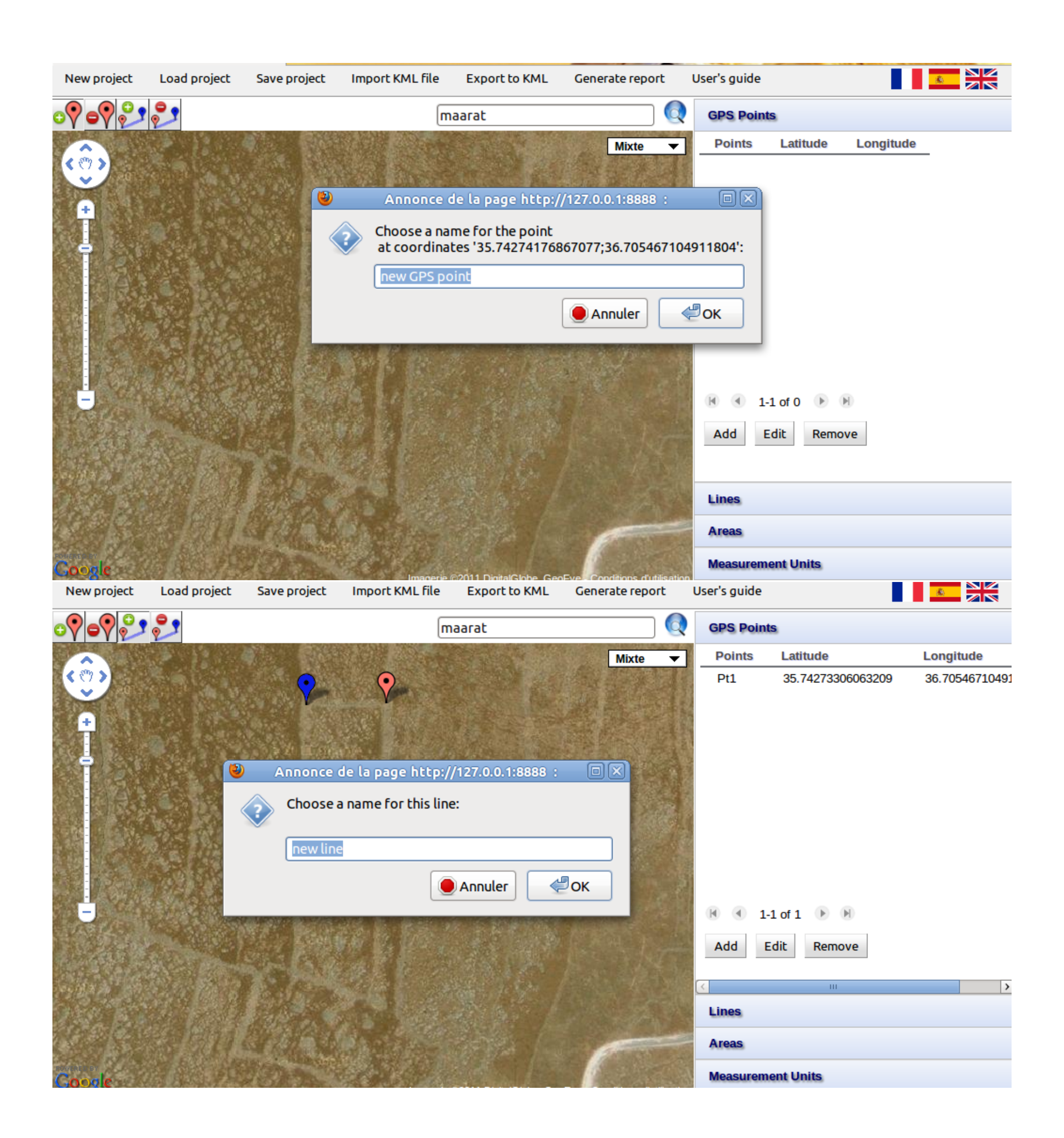


Fig. 3.1 – Vue sur la carte

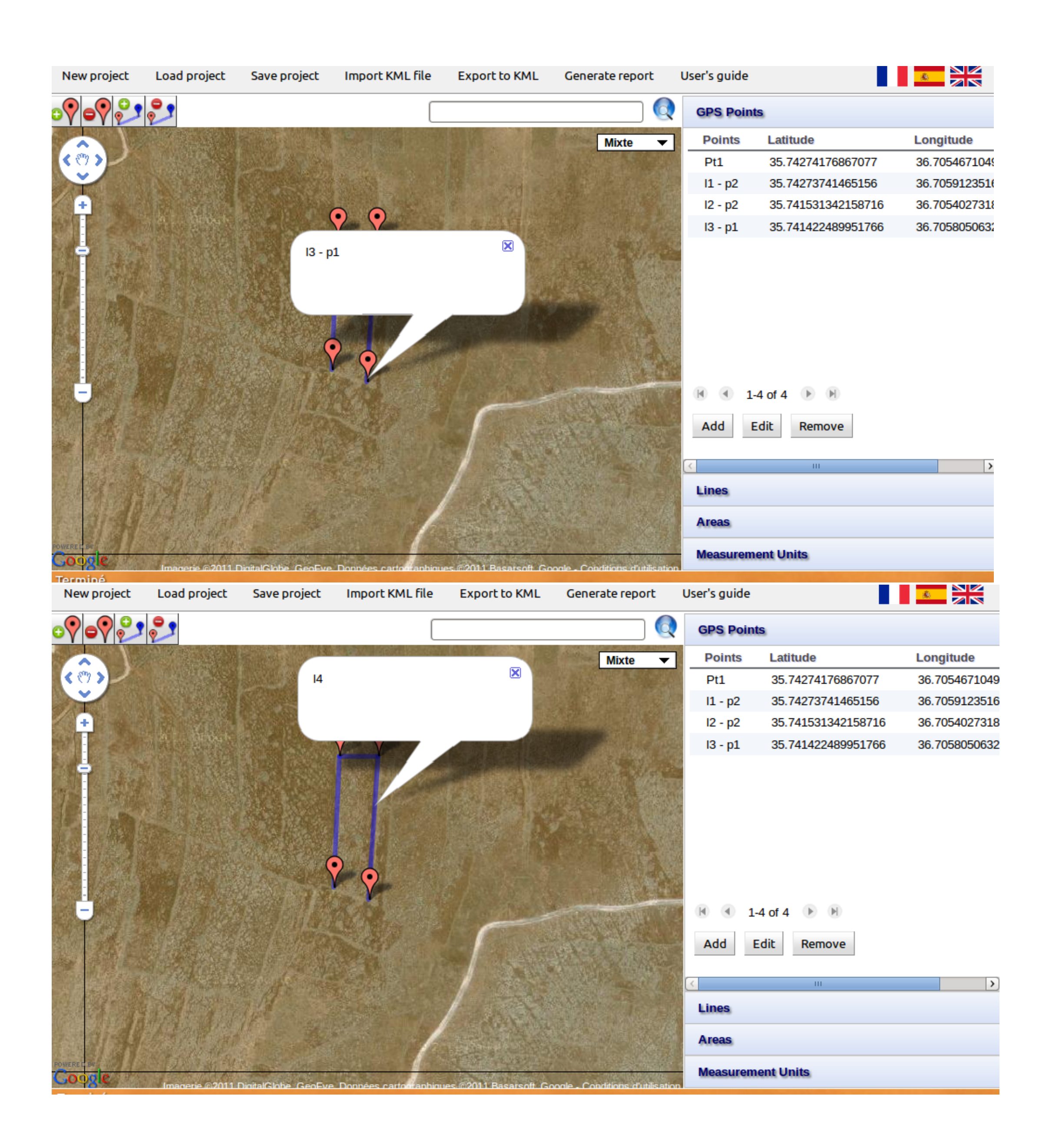
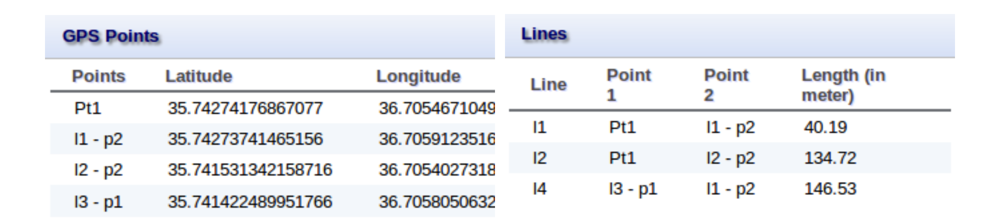


Fig. 3.2 – Vue sur la carte



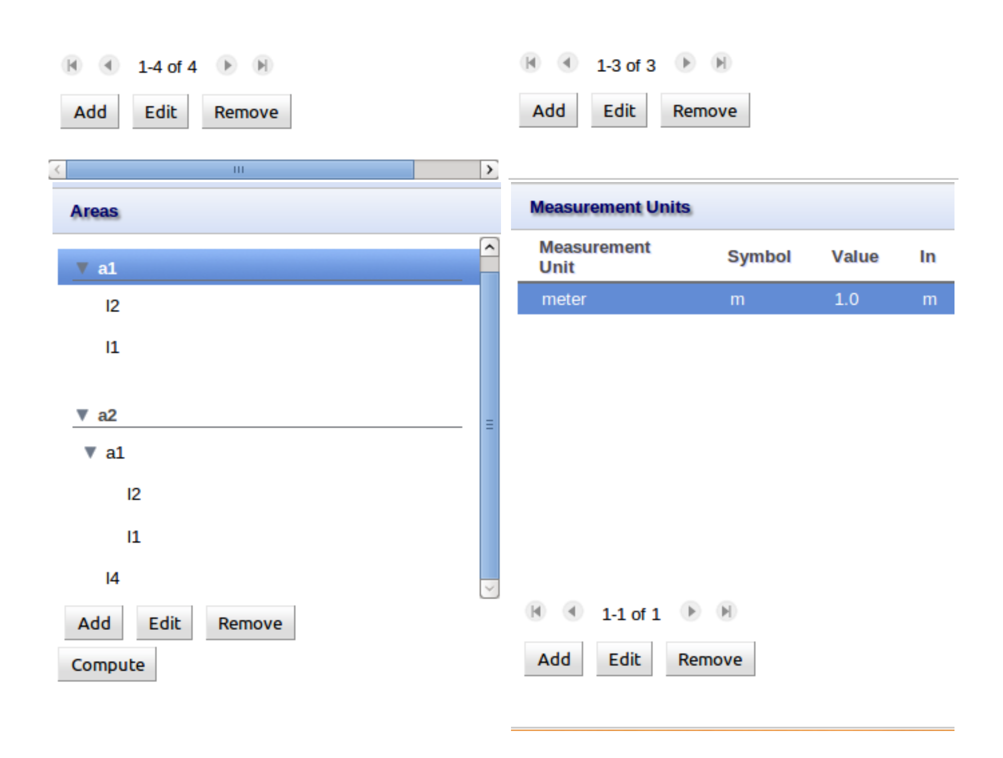


FIG. 3.3 – Les panneaux d'informations

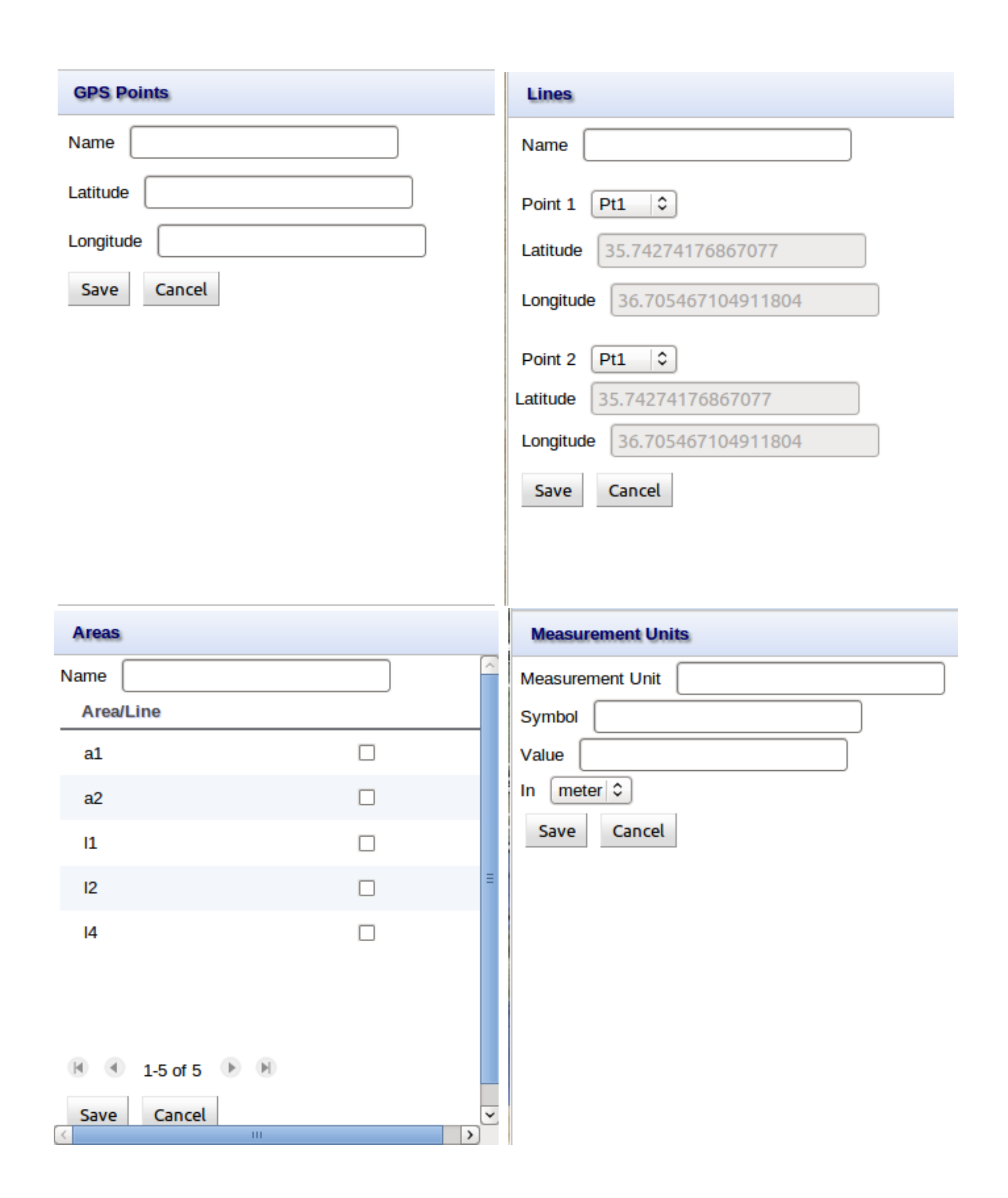
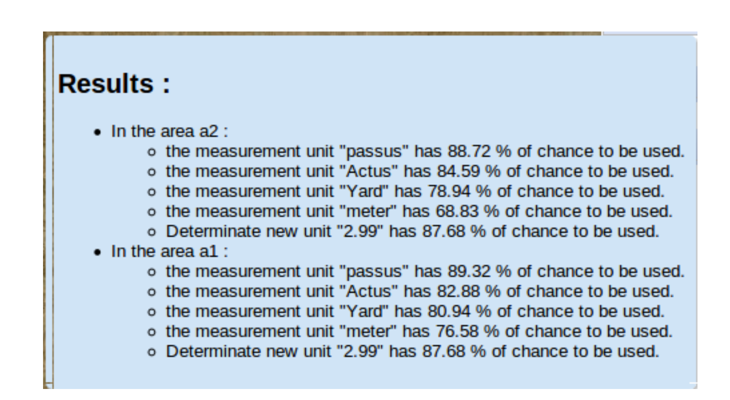


FIG. 3.4 – Ajout et édition des entités


Results :

- In the area a2 :
  - the measurement unit "passus" has 88.72 % of chance to be used.
  - the measurement unit "Actus" has 84.59 % of chance to be used.
  - the measurement unit "Yard" has 78.94 % of chance to be used.
  - the measurement unit "meter" has 68.83 % of chance to be used.
  - Determinate new unit "2.99" has 87.68 % of chance to be used.
- In the area a1 :
  - the measurement unit "passus" has 89.32 % of chance to be used.
  - the measurement unit "Actus" has 82.88 % of chance to be used.
  - the measurement unit "Yard" has 80.94 % of chance to be used.
  - the measurement unit "meter" has 76.58 % of chance to be used.
  - Determinate new unit "2.99" has 87.68 % of chance to be used.

Fig. 3.5 – PopUP des resultats

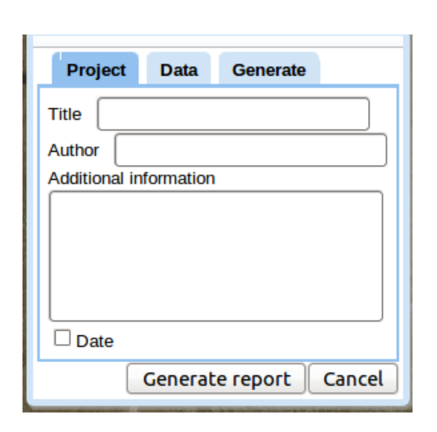


Fig. 3.6 – Informations sur le rapport.

L'entête

Cette étape permet d'indiquer les informations de l'entête du rapport(voir fig. 3.6) (titre, auteur, notes, date).

Le contenu

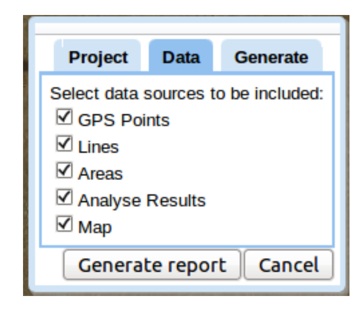


FIG. 3.7 – Le contenu du rapport.

Permet d'insérer ou d'enlever des parties du rapport (voir fig. 3.7).

Le format

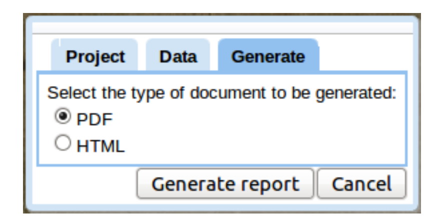


FIG. 3.8 – Format du rapport.

Choisir le format de sortie (HTML/PDF) (voir fig. 3.8).

Chapitre 4

Architecture

4.1 Modèle MVP

4.1.1 Introduction au modèle MVP

Le modèle MVP (Model View Presenter, voir fig. 4.1) est un modèle d'architecture dérivé du modèle MVC (Model View Controler).

Le principe est simple, l'architecture du programme est dévisée en 3 :

- Modèle : les modèles représentent les données de l'application.
- Vue : Les vues représentent des ensembles d'éléments graphiques. Par exemple, dans notre logiciel, la carte GoogleMap, la liste des points GPS, la liste des lignes représentent chacun une vue.
  Une vue implémente une interface définie par le présenteur, qui lui est associée, et ainsi l'implémentation de l'interface graphique est indépendante du présenteur.
- Présenteur : Un présenteur est associé à chaque vue. Son rôle est de gérer la synchronisation de la vue avec le serveur et de gérer tous les évènements provenants de widgets internes à la vue à laquelle il est associé.
  Chaque présenteur décrit une interface décrivant les actions qui sont associées à la vue. De cette façon le présenteur ne sait pas comment est implémentée la vue.

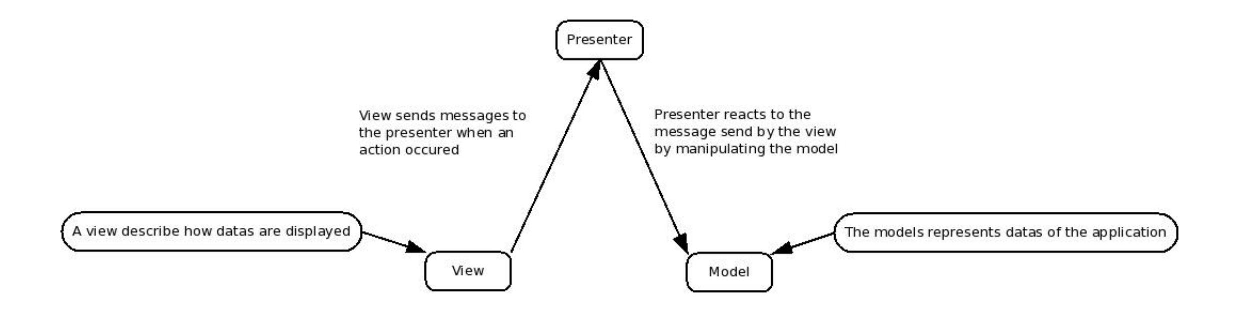


FIG. 4.1 – Modèle MVP

4.1.2 Extension du modèle

Dans le cas de notre application, l'architecture doit prendre en compte le fait que le logiciel est devisé en une partie client et une autre serveur.

Les vues et présenteurs seront exécutées dynamiquement coté client, et les présenteurs feront des appels asynchrones au serveur pour manipuler les modèles.

L'interaction entre les modèles, les vues, les présenteurs et le serveur sera détaillée dans les diagrammes de séquences à la fin de ce chapitre.

4.1.3 Avantages de ce modèle

Ce modèle de conception apporte deux avantages majeurs.

Le premier est qu'il permet de séparer les fonctionnalités et comportement de notre logiciel de façon à rendre indépendants les données, l'interface graphique et le traitement des évènements et à faciliter le travail collaboratif.

Le second avantage par rapport au modèle MVC et qu'il ajoute une couche supplémentaire entre les modèles et les vues (les présenteurs). L'utilisation de présenteurs permet de simplifier l'interaction entre les vues et les modèles et permet de modifier l'interface graphique sans avoir à modifier autre chose dans le code.

4.2 Diagrammes de paquetages

4.2.1 Hiérarchique

Ce diagramme de paquetage (voir fig. 4.2) montre le découpage de notre architecture en présentant la hiérarchie des paquetages. Le contenu de chaque paquetage est détaillé plus loin.

Une application GWT classique est séparée en trois paquetages :

- Un paquetage client, dont les classes seront compilées en JavaScript et exécutées par le navigateur.
- Un paquetage server, qui sera compilé en bytecode Java, et exécuté par un serveur d'applications.
- Un paquetage shared, qui contiendra toutes les classes utilisées à la fois par le client et le serveur.

4.2.2 Relationnel

Ce diagramme de paquetage (voir fig. 4.3) montre la relation UML "import" entre les différents paquetages, c'est-à-dire les relations d'utilisation entre les paquetages.

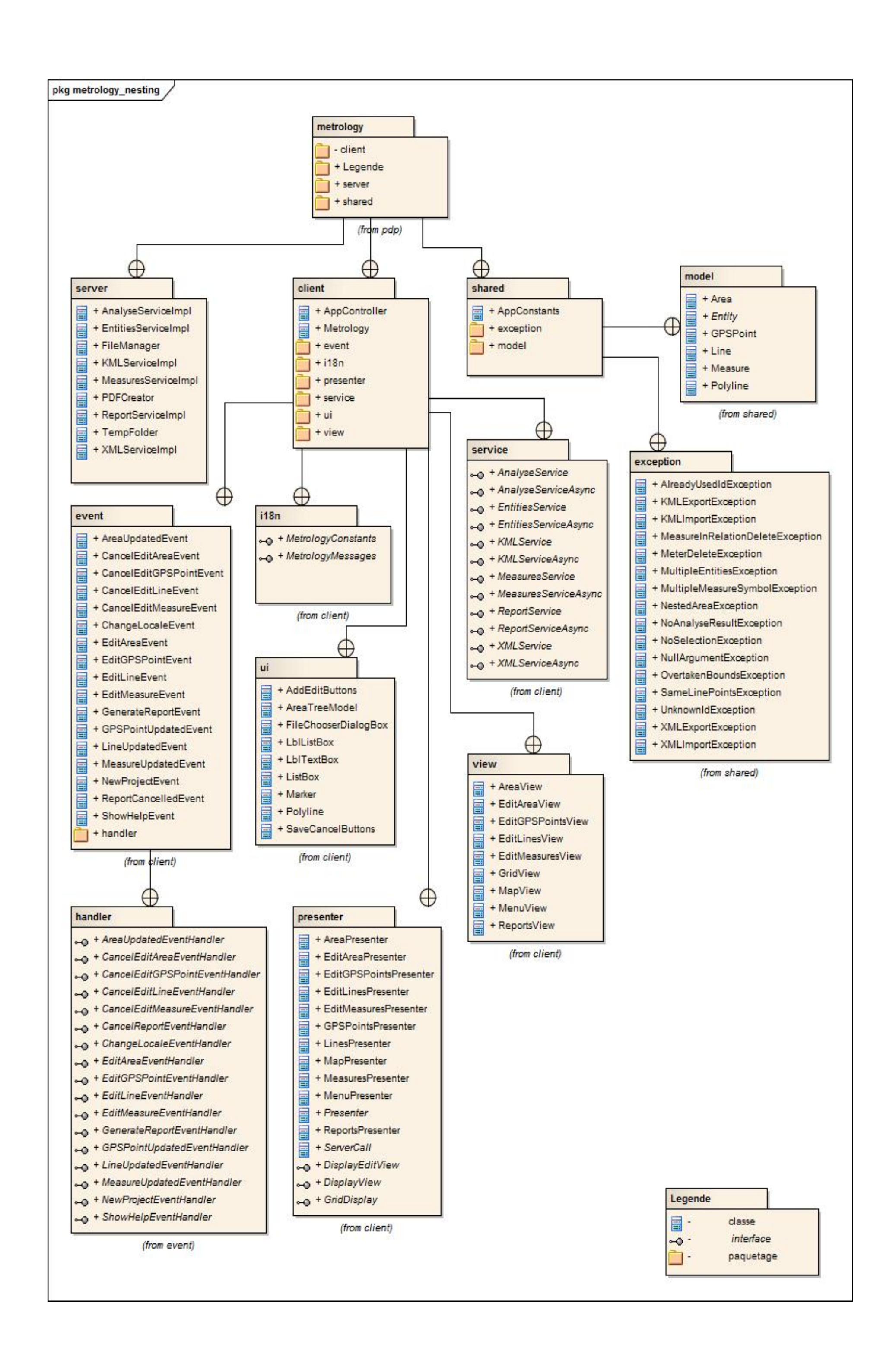


FIG. 4.2 – Diagramme de paquetages hierarchique

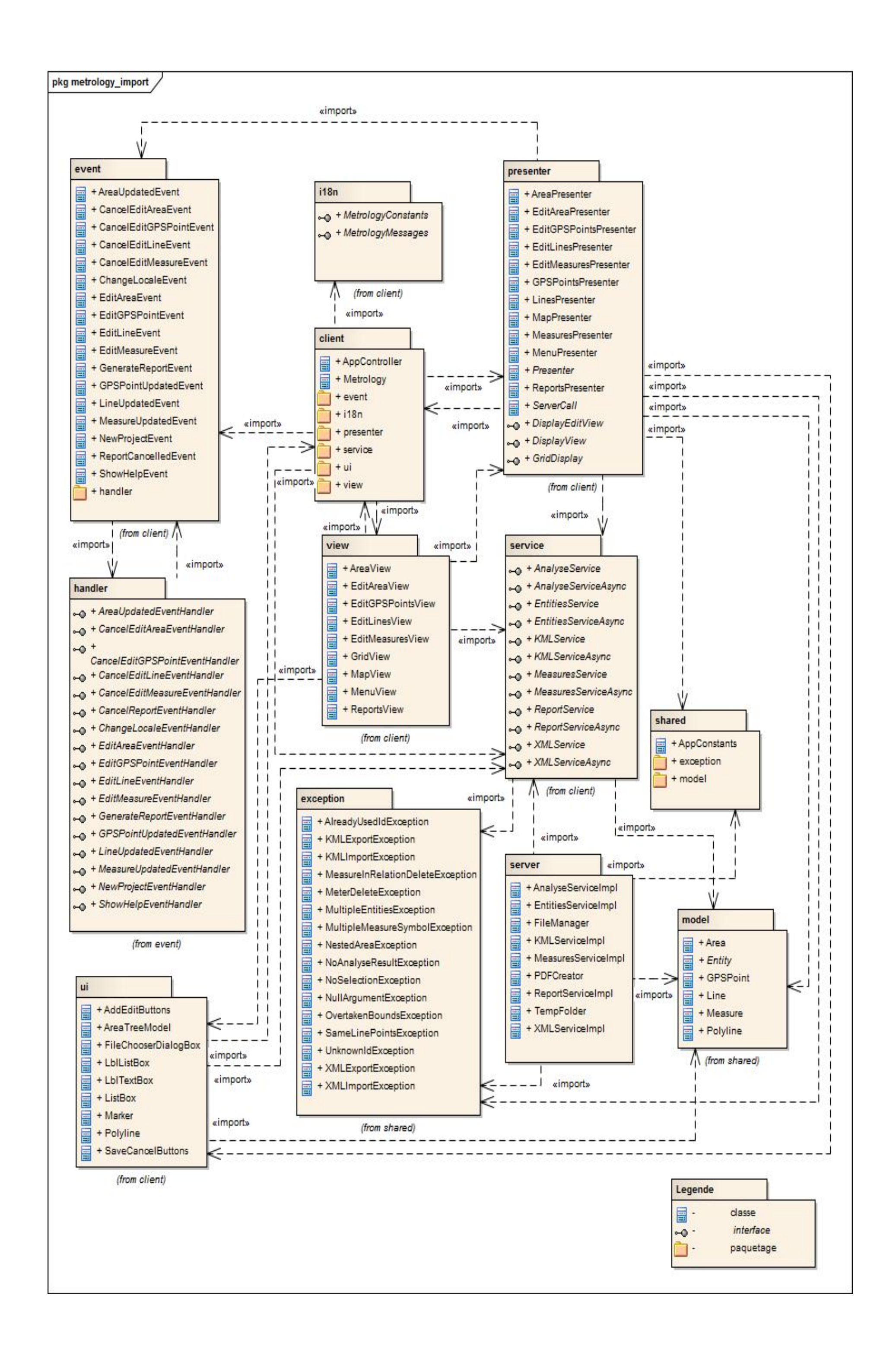


FIG. 4.3 – Diagramme de paquetages relationnel

4.3 Diagrammes de classes

4.3.1 shared.model

Ce paquetage contient les modèles, c'est-à-dire les données du logiciel. Nous avons deux sortes de données, les entités que l'on ajoute sur une carte et les unités de mesure.

La classe abstraite *Entity* contient les informations communes aux classes *GPSPoint*, *Line* et *Area*. Une *Area* est un ensemble de *Line* et *Area*.

La classe *Measure* représente une unité de mesure, elle contient son nom et sa valeur dans une autre unité de référence.

Toutes ces classes implémentent l'interface *IsSerializable* (variante made-in GWT de *Serializable*), pour permettre aux objets d'être transférés entre le client et le serveur.

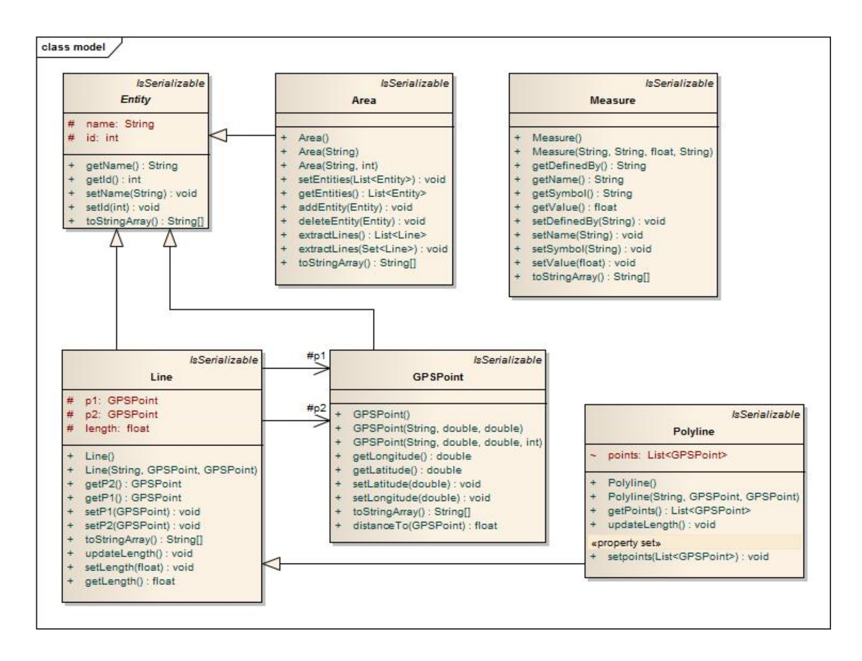


Fig. 4.4 – Diagramme de classes, paquetage *shared.model*

4.3.2 *shared.exception*

Ce paquetage contient les différentes exceptions utilisées par notre application, que nous avons héritées de *java.lang.Exception*

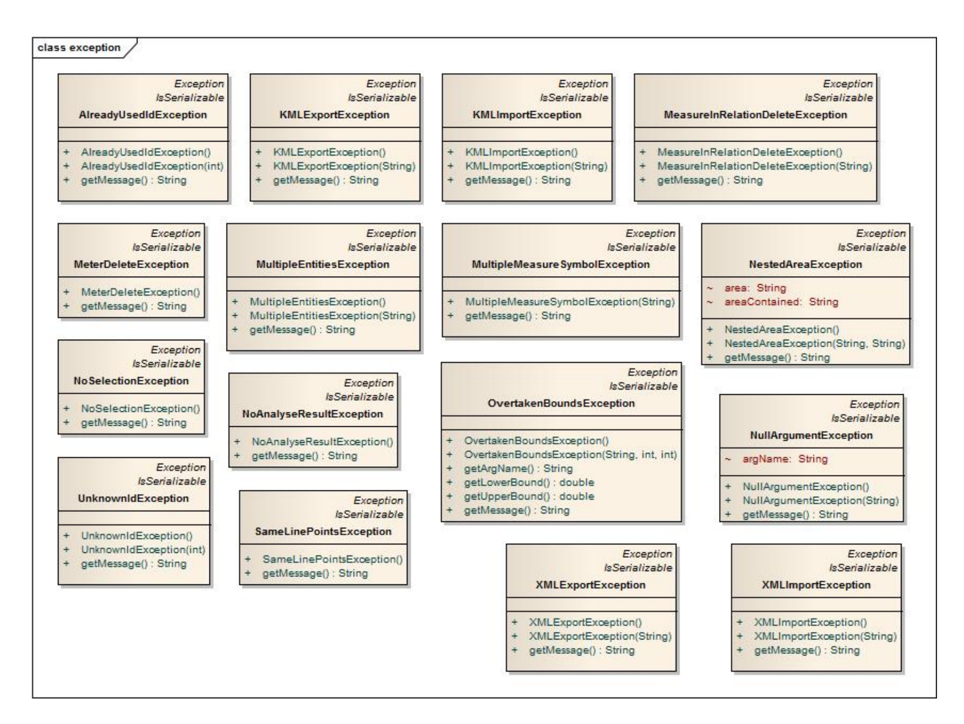


Fig. 4.5 – Diagramme de classes, paquetage *shared.exception*

4.3.3 client

Ce paquetage contient deux classes :

1. **AppController** C'est cette classe qui se charge de créer les différents présenteurs et vues, et qui place ces dernières dans les différents conteneurs qui séparent la fenêtre du navigateur. C'est également cette classe qui se chargera de rafraîchir ces vues lorsque les modèles seront modifiés.
2. **Metrology** Point de lancement de l'application (elle implémente pour cela la classe *EntryPoint*), c'est cette classe qui crée l'objet *AppController*. Un certain nombre de variables statiques utilisées coté client seront dans cette classe.

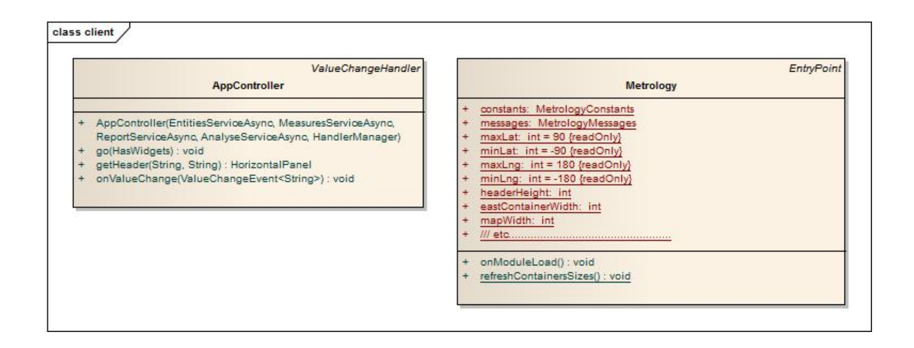


FIG. 4.6 – Diagramme de classes, paquetage *client*

4.3.4 *client.view*

Dans ce paquetage sont contenues toutes les vues. Chaque vue hérite de la classe *Composite* de GWT, et est donc un ensemble de widgets. Chaque vue implémente également une interface *Display*, qui sera utilisée par le(s) présenteur(s) correspondant. Ces interfaces permettent aux présenteurs de contrôler les vues sans connaître le type d'objets utilisées.

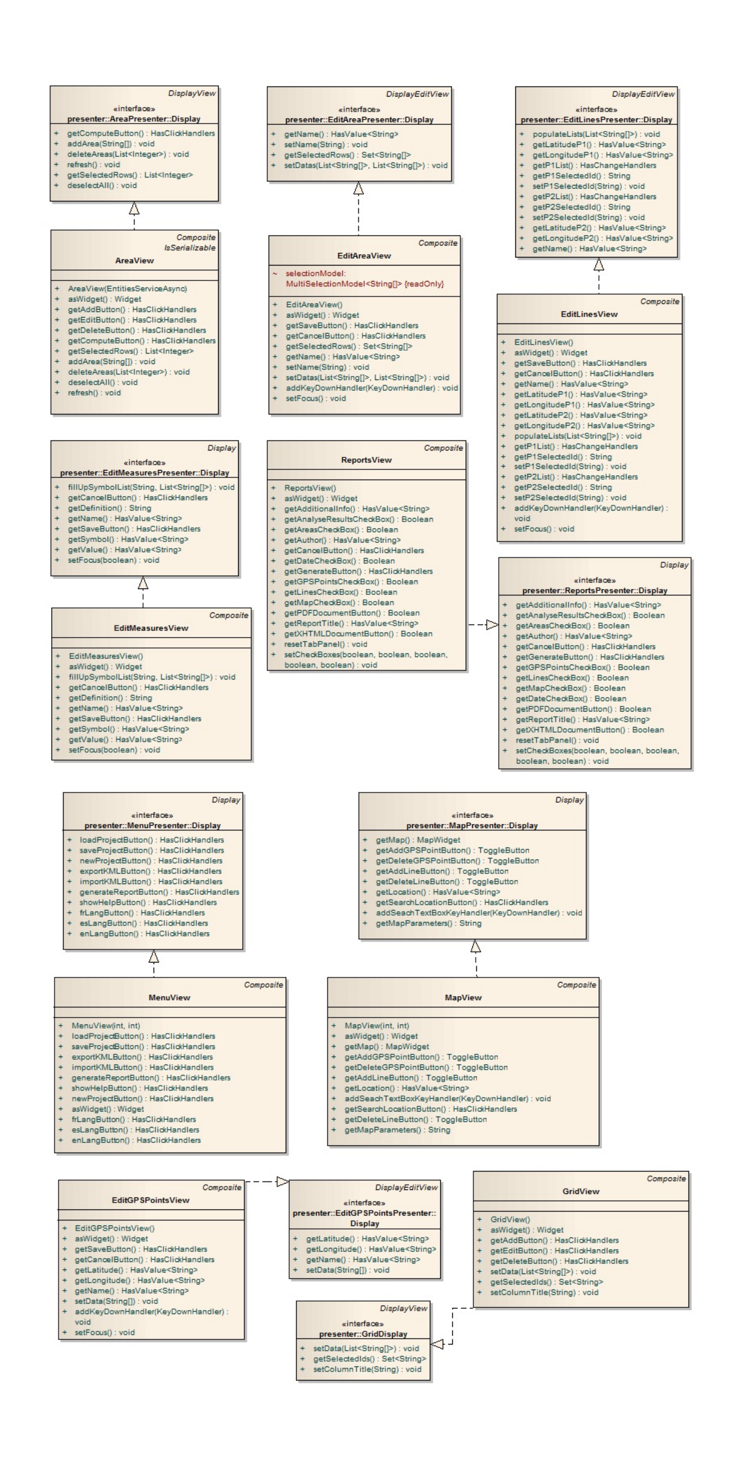


FIG. 4.7 – Diagramme de classes, paquetage *client.view*

4.3.5 *client.presenter*

Ce paquetage (voir fig. 4.8)contient les présenteurs, ce sont eux qui "donnent vie" aux vues, en associant des actions aux boutons, etc, comme dis ci-dessus. Chaque présenteur contient une interface qui hérite de *Display*, implémentée par une vue. De cette façon, le présenteur sait quels éléments sont dans chaque vue, sans savoir exactement quels widgets sont utilisées.
Ce sont les présenteurs qui réalisent les appels asynchrones au serveur, pour obtenir les modèles (points GPS, lignes, etc...)
Ce sont également ces présenteurs qui informent l'*AppController* qu'un modèle à été modifié, pour que les vues concernées soient actualisées.

4.3.6 *client.event*

Pour qu'un présenteur informe les autres qu'il a modifié un modèle, il lance un evènement de ce paquet (voir fig. 4.9). De cette façon, un presenter ne s'occupe pas de quel objet traite ces évenements.

4.3.7 *client.event.handler*

Ce paquetage (voir fig. 4.10) regroupe les handlers des évènements du paquetage *client.event*.

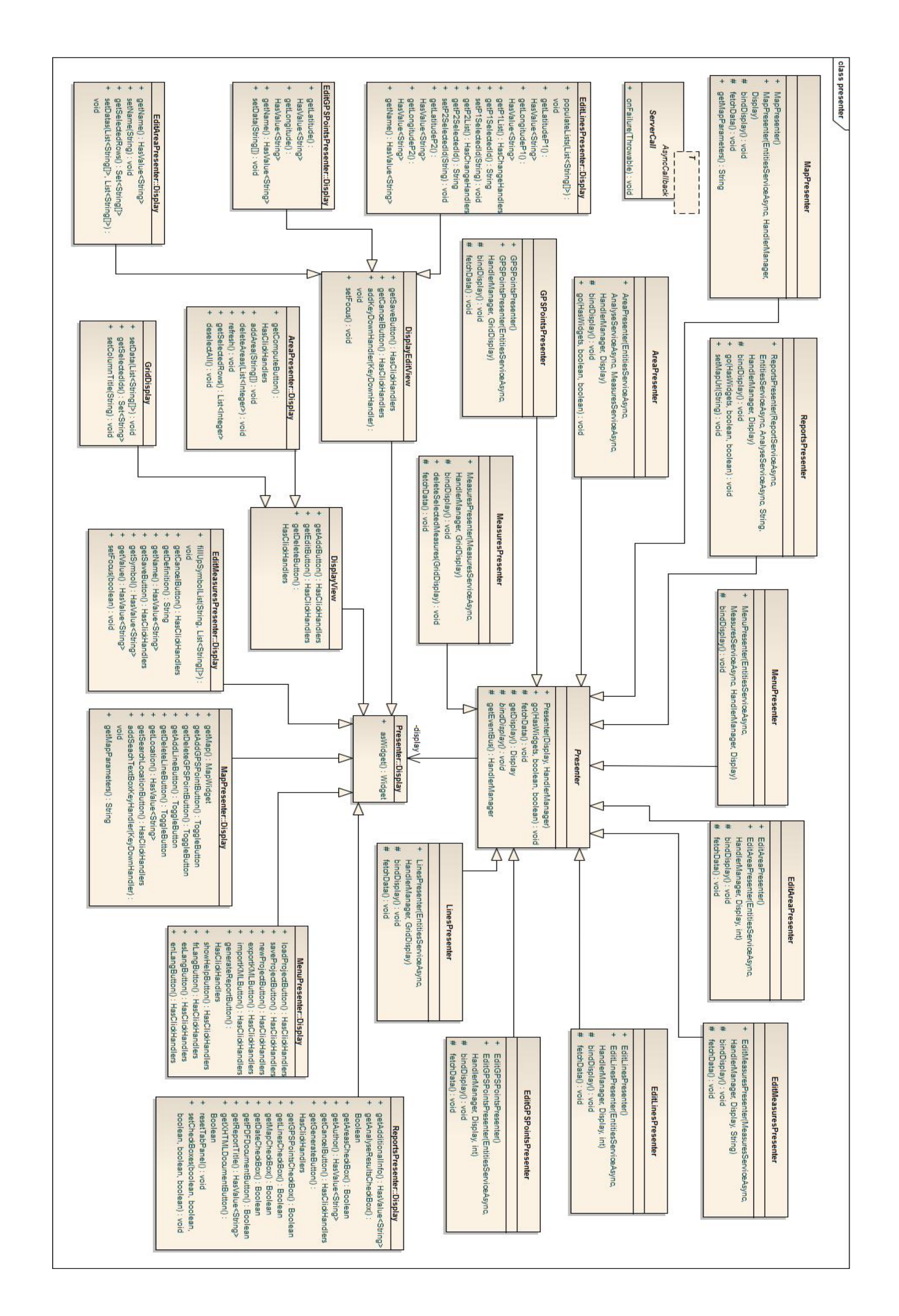


FIG. 4.8 – Diagramme de classes, paquetage client.presenter

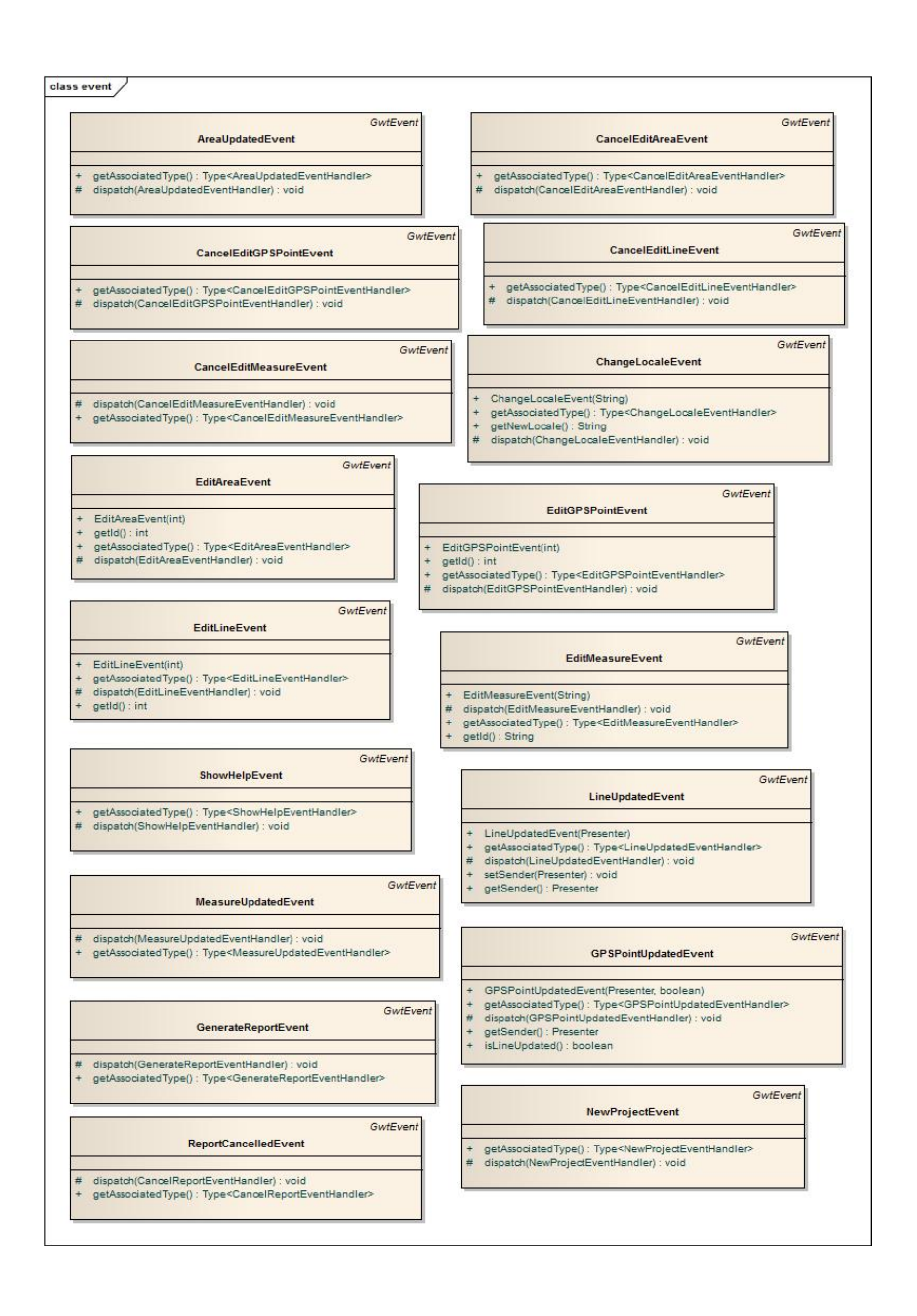


FIG. 4.9 – Diagramme de classes, paquetage client.event

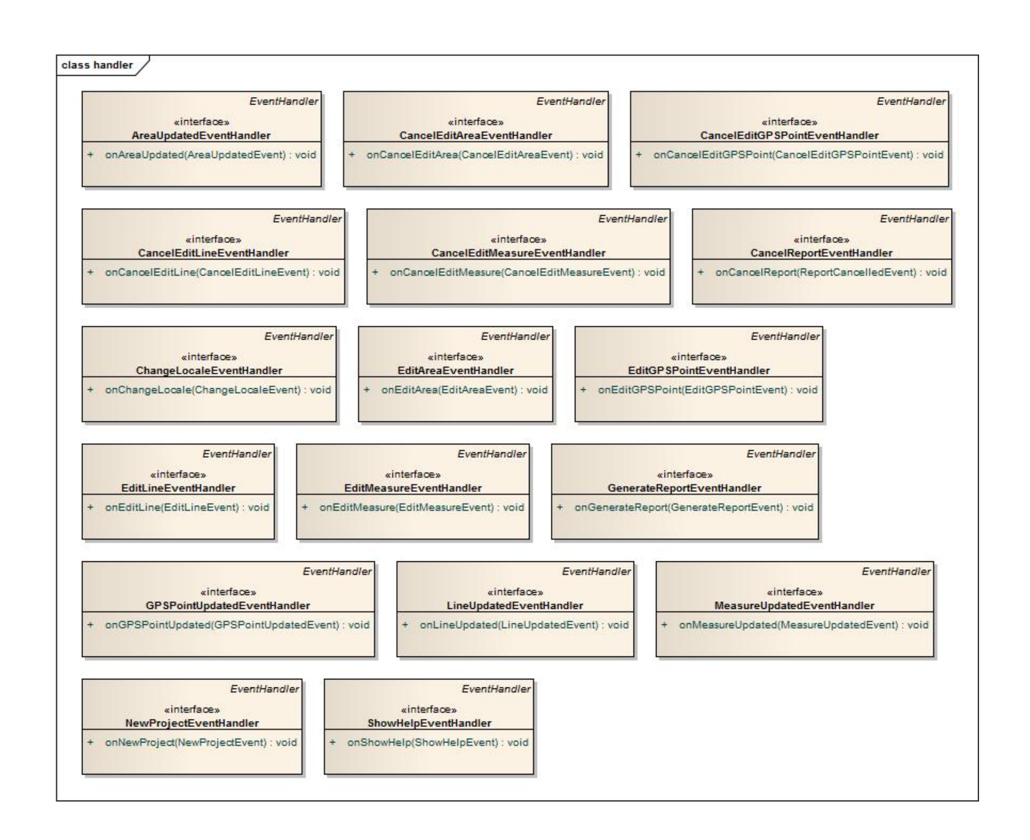


Fig. 4.10 – Diagramme de classes, paquetage *client.event.handler*

4.3.8 *client.service*

Ce paquetage (voir fig. 4.11) regroupe les interfaces des différents services. On constate la présence de classes **Service* et **ServiceAsync* : ceci est une convention de GWT : les interfaces **Service* seront implémentées coté serveur, et les **ServiceAsync* permettent au coté client de réaliser des appels asynchrones au serveur.

1. ***EntitiesService* :** gère l'ajout, suppression et édition des points GPS, lignes, et zones.
2. ***MeasuresService* :** gère l'ajout, suppression et édition des unités de mesures.
3. ***KMLService* :** définit les méthodes permettant d'importer/exporter des fichiers KML.
4. ***XMLService* :** définit les méthodes permettant de charger/sauvegarder un projet, au format XML.
5. ***AnalyseService* :** gère le calcul des unités de mesures utilisées dans une zone définie par l'utilisateur.
6. ***ReportService* :** génère des rapports dans différents formats (dont le PDF), pour permettre à l'utilisateur de sauvegarder les résultats renvoyés par l'AnalyseService.

4.3.9 *server*

Dans ce paquetage (voir fig. 4.12) se trouvent les implémentations des différents services du paquetage client.service, ainsi que les servlets suivants :

1. ***FileManager* :** gère les téléchargements des fichiers entre le client et le serveur. Ce servlet sera appelé lors du chargement et de la sauvegarde de fichiers.
2. ***PDFCreator*** génère un fichier PDF à partir d'une page Html, ce servlet est appelé lors de la génération du rapport.

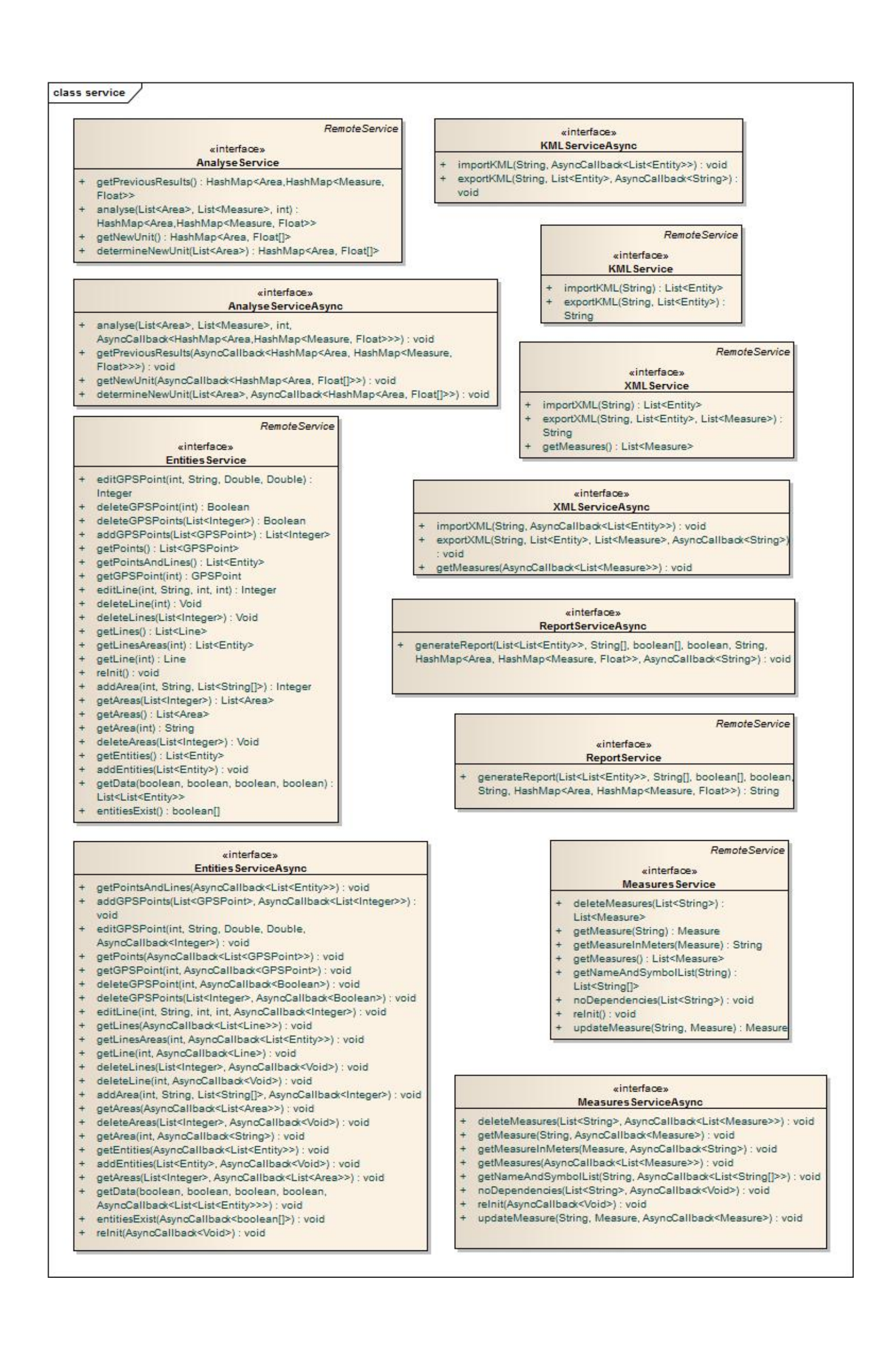


FIG. 4.11 – Diagramme de classes, paquetage *client.service*

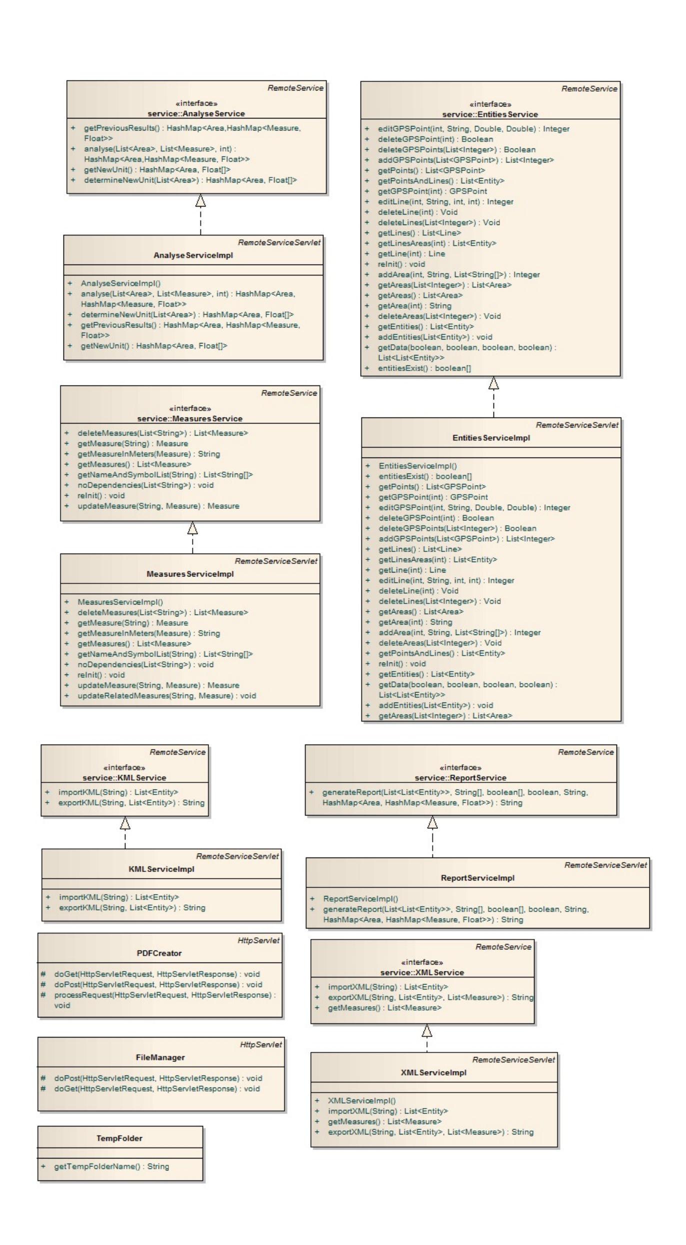


FIG. 4.12 – Diagramme de classes, paquetage *server*

4.3.10 *client.ui*

Ce paquetage regroupe les widgets que nous avons créés pour permettre de réaliser plus rapidement la réalisation des différents vues.

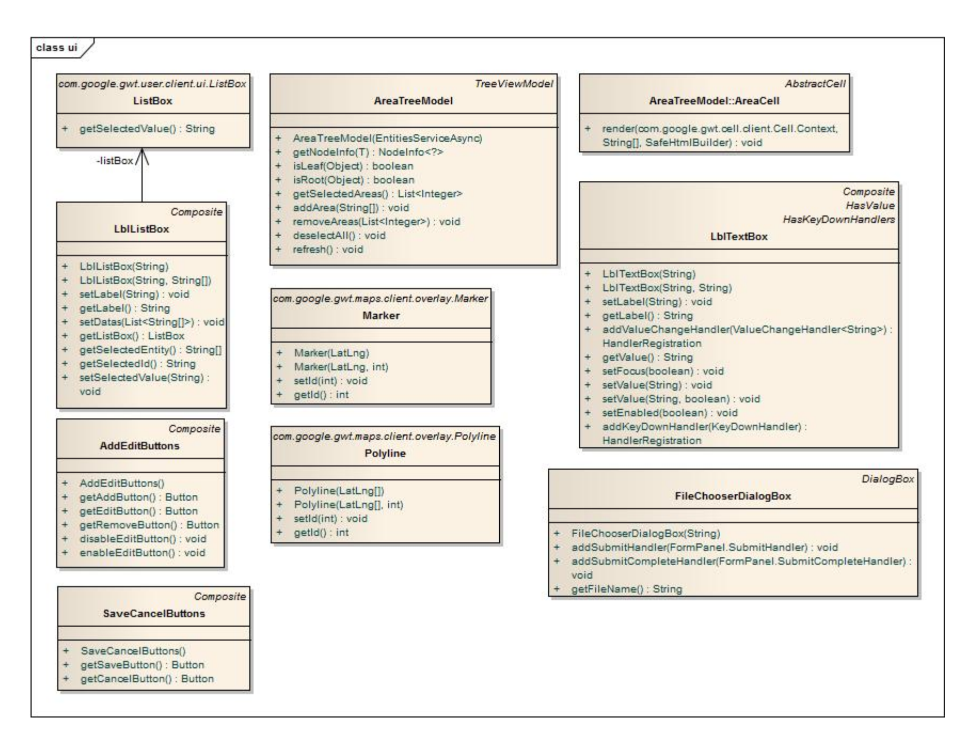


FIG. 4.13 – Diagramme de classes, paquetage *client.ui*

4.3.11 *client.i18n*

Le paquetage *i18n* (pour internationalisation) sert à l'internationalisation de notre application. Les deux interfaces MetrologyConstants et Metrology-Messages définissent respectivement les chaines constantes et celles prenant des paramètres.

A chaque langue est associé deux fichiers textes contenants les traductions des chaines et c'est à l'exécution que le choix de la chaine à afficher se fait, en fonction de la langue choisie.

La langue par défaut de l'application est l'anglais.

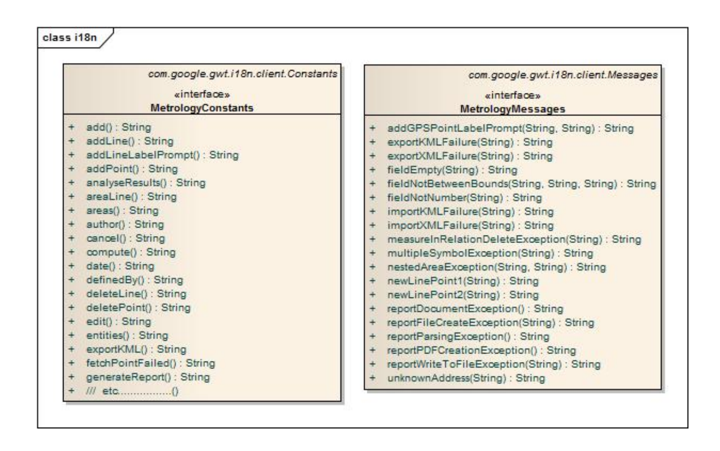
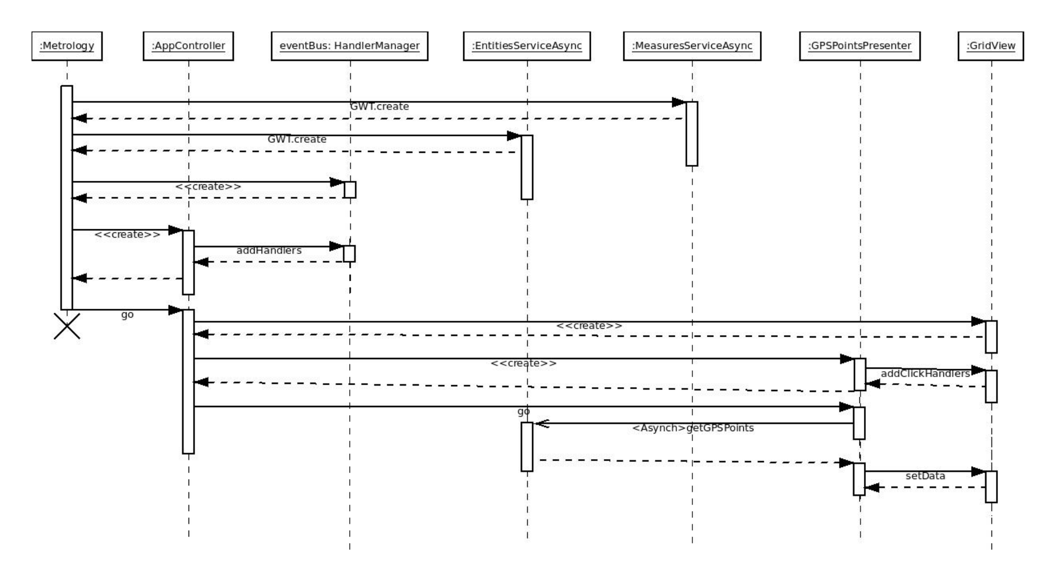


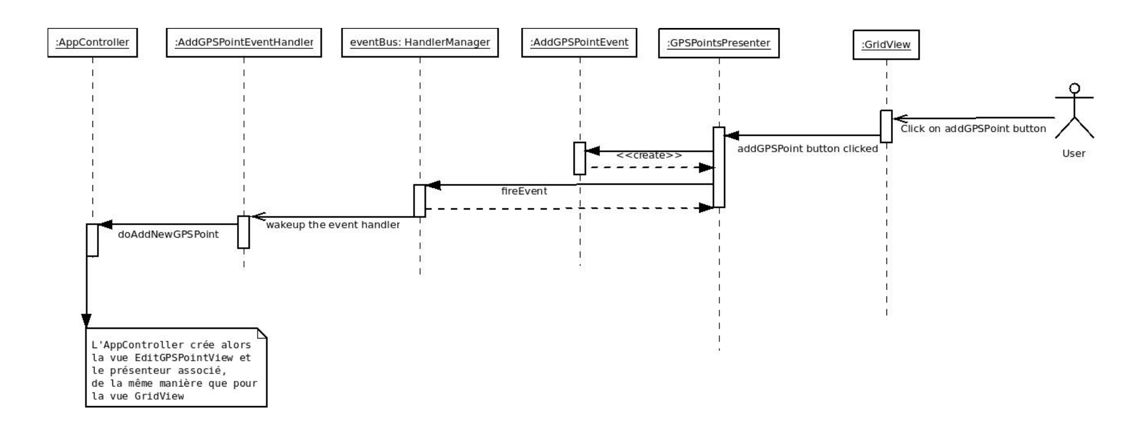
FIG. 4.14 – Diagramme de classes, paquetage *client.i18n*

4.4 Diagrammes de séquence



Ce diagramme décrit le lancement de l'application, la création des objets de la classe *AppController*, *EventHandler*, la création des services qui doivent être disponible au démarrage de l'application et l'affichage de la première vue.

Pour simplifier nous avons choisi de montrer uniquement l'affichage de la vue de la liste des points GPS et la création des services associés (*Entities-Service*).



Ce diagramme décrit comment l'application réagit au clique de l'utilisateur sur un bouton qui demande l'affichage d'une nouvelle vue. Nous avons pris comme exemple le clique sur le bouton pour ajouter un nouveau point GPS.

Chapitre 5

Choix d'implémentations

5.1 Choix du langage utilisé

Une contrainte importante dans ce choix à été la présence d'une api, dans ce langage, qui permette l'intégration de Google Maps dans un navigateur. Il se trouve qu'il existe une api liée à GWT, permettant de faire ceci. GWT est un ensemble d'outils développés par Google permettant de compiler du code Java en JavaScript, exécutable par un navigateur, et permet une interaction client-serveur via des messages RPC (*Remote Procedure Call*). Google a aussi développé un plugin GWT pour Eclipse, ce qui a défini directement notre IDE.

5.2 Choix algorithmiques

5.2.1 Calcul des distance

Les distances des lignes sont calculées en géométrie euclidienne, en utilisant la latitude et la longitude de chaque point. Il existe plusieurs méthodes permettant d'approximer la distance entre deux points GPS. Les deux principales sont d'utiliser la distance du grand cercle et la distance euclidienne. Dans la première formule, la terre est approximée à une sphère et la distance calculée correspond à la plus petite distance entre deux points autour de cette sphère.

$$Dgc = 2R \arcsin \sqrt{\sin^2\left(\frac{lat2-lat1}{2}\right) + \cos(lat1)\cos(lat2)\sin^2\left(\frac{long2-long1}{2}\right)}$$

où R est le rayon de la terre, ($lat1$, $long1$) et ($lat2$, $long2$) sont respectivement la latitude et longitude des deux points.

La seconde formule, permet de calculer la distance entre deux points dans un espace euclidien. Le résultat obtenu est donc la distance en ligne droite entre les deux points.

$Deucl = R * \sqrt{\sin^2(lat2 - lat1) + \cos(lat1) * \cos(lat2) * \sin^2(lon2 - lon1)}$

Étant donné, que les distances mesurées sont relativement petites (moins d'un kilomètre), l'approximation euclidienne est suffisante pour obtenir une erreur inférieure à 1 mètre pour 20 mètres.

en effet, si l'on considère les points GPS (35.741992,36.703734) et (35.740730,36.703348).
$Dgc = 144,6m$
$Deucl = 144,67m$

Dans cet exemple, l'erreur est inférieure à $10^{-1} mètre, lesdistancescalculéesétantdeslongueurs$

5.2.2 Calcul de la probabilité d'utilisation d'une unité de mesure

Ce calcul est réalisé en faisant une moyenne des modulo entre la longueur de chaque ligne et l'unité de chaque ligne (en réalité nous prenons pour modulo le minimum entre (longueur)%(unité de mesure) et (unité de mesure)-(longueur)%(unité de mesure)). Ensuite, nous divisons cette moyenne par l'unité de mesure, et nous multiplions par cent pour obtenir la probabilité d'utilisation de l'unité de mesure dans la zone concernée.

5.2.3 Détermination d'une nouvelle mesure

L'algorithme utilisé pour déterminer une mesure ayant pu être utilisé est simple, pour chaque zone, pour chaque lignes de la zone, nous divisons la longueur de la ligne par le nombre possible de répétition de l'unité de mesure. Ensuite le calcul précédent est appliqué pour toutes les lignes de la zone.

5.3 Choix techniques

5.3.1 La classe Entity

la classe abstraite *Entity* est la classe dont dérivent les modèles *GPS-point*, *Line* et *Area*. cette classe permet de définir les méthodes et attributs communs à ces classes.

5.3.2 Définition d'un id pour chaque entité

Pour accéder rapidement à une entité (les objets qui héritent de *Entity*, et les *Measure*), nous avons choisi, plutôt que de mettre tous ces objets dans une liste et de les comparer avec la méthode *equals()*, d'ajouter un id (un entier pour les *Entity*, le symbole de l'unité de mesure pour les *Measure*) à chaque objet, et de les stocker sur le serveur dans des *HashMap* (ayant pour clé l'id de l'objet, et pour valeur l'objet lui-même). L'id des *Entity* est généré automatiquement, via une simple incrémentation d'un compteur dans la classe *EntitiesServiceImpl*, et l'id des *Measure* est défini par l'utilisateur (l'unicité de cet id est vérifié lors de l'ajout ou de l'édition d'une unité de mesure).

5.3.3 Les unités de mesure

Le logiciel permet à l'utilisateur de gérer les unités de mesure. Chaque unité de mesure est représentée par une instance de la classe *Measure*. Une unité de mesure valide doit avoir un nom, un symbole, une valeur et elle doit être définie par une autre unité de mesure. Les unités de mesure sont gérés par le *MeasureServiceImpl* servlet. Ce service implémente une HashMap qui contient les unités de mesure. L'avantage de l'utilisation d'une HashMap est qu'en utilisant le symbole de l'unité comme clé, nous sommes sûr de l'unicité de ce symbole. Comme chaque nouvelle unité de mesure doit être défini par une autre, il doit y avoir une unité de mesure par défaut. Dans notre cas, l'unité de mesure par défaut est le mètre. Le mètre est de cette façon "intouchable". Il n'est pas possible de modifier ou de supprimer cette unité de mesure. Il y a aussi des restrictions lorsqu'il s'agit de mesures définies par l'utilisateur. L'utilisateur ne peut pas supprimer une unité de mesure qui définit une autre unité de mesure. Lorsque l'utilisateur modifie une unité de mesure le changement va être appliqué à toutes les unités de mesure qui sont définies par l'unité qui a été modifié.

5.3.4 La vue *GridView*

Après quelques temps passé à développer indépendamment les différentes vues qui affichent respectivement les points GPS, les lignes, et les unités de mesure, nous avons remarqué que celles ci étaient en tout point identiques, à ceci près qu'elles n'utilisaient les mêmes types d'objets. Pour remédier à cela, nous avons ajouté à nos modèles une méthode *toStringArray()*, qui renvoie les informations affichées sous forme d'un tableau de String, l'élément 0 étant dans tous les cas l'id de l'objet. La seule différence restante entre

ces vues étant le nombre de colonnes, il a suffit d'ajouter la méthode *setColumnTitle(String... titles)*, qui prend donc un nombre de paramètres variable, définissant le nombre de colonnes affichées.

Cette factorisation des vues est possible grâce à l'architecture MVP de notre application. Nous avons donc ici tiré profit du modèle de conception en utilisant l'indépendance entre les vues et les modèles pour factoriser et réutiliser du code.

5.3.5 Représentation des zones

Nous avons choisi, pour afficher les zones d'utiliser un arbre. Cet arbre est constitué de nœuds qui représentent les zones et de feuilles qui représentent les lignes et autres zones. L'utilisation d'une arborescence permet de se déplacer très rapidement dans les zones incluant de nombreuses autres zones. L'implémentation est cet arbre est décrite dans la classe *AreaTreeModel* qui est une extension de la classe *TreeViewModel*. La classe TreeModel défini une méthode *getNodeInfo*, qui est appelé pour chaque noeud de l'arbre et qui défini le traitement associé à chaque niveau. Le premier niveau correspond à la racine de l'arbre, représentée par un noeud null. Le traitement associé est de récupérer, via un appel au serveur ,la liste de toutes les zones et de les inclure dans l'arbre. Le second niveau correspond au contenu d'une zone. Dans ce cas, il faut appeler la méthode getLinesArea(int id) qui retourne la liste des entités contenues dans la zone. Cette implémentation est cependant problématique par rapport au choix de noter architecture. En effet, l'utilisation de TreeViewModel impose que la classe effectue des appels synchrone pour remplir l'arbre ou, si l'on souhaite utiliser des appels asynchrones (comme c'est la cas dans notre application client-serveur), impose le fait d'utiliser directement les appels au serveur, sans pouvoir passer par un présenteur. Ceci est du au fait que la classe *TreeViewModel* ne possède pas de moyen de rafraichissement en cas de modification externe de ses données. Malgré cela, nous avons décidé d'utiliser un *TreeViewModel* car aucun des autres widgets présents dans Gwt ne permet d'afficher d'une manière efficace les zones.

5.3.6 Synchronisation de la carte et des listes

Les points et lignes peuvent, comme vu dans l'exemple d'utilisation, être ajoutés ou modifiés directement depuis la carte, ou depuis les listes du panneau de droite. C'est ici que le mécanisme des *Event* est intéressant : lorsqu'une de ces entités est modifiée, le présenteur associé à la vue qui la modifie

va envoyer un évènement sur le bus d'évènements (via la méthode *eventBus.fireEvent()*) et cet évènement va être attrapé par l'objet *AppController*. Selon le type du présenteur qui a envoyé l'évènement, l'*AppController* va ensuite appeler les présenteurs concernés pour leur faire recharger la liste des entités.

5.3.7 La classe *ServerCall*

La classe *ServerCall* est une classe abstraite que nous avons créé qui implémente l'interface *AsyncCallBack*. Cette interface contient deux méthodes *OnSuccess()* et *OnFailure()* qui sont appelé lors du retour d'un appel asynchrone au serveur. L'usage en GWT est de pour chaque appel asynchrone de définir une classe interne qui implémente ces deux méthodes. Cependant, lors de tous nos appels au serveur, en cas d'échec, une exception est lancée qui contient le message d'erreur à afficher à l'utilisateur, le traitement à effectuer est donc toujours le même. Nous avons donc créé la classe *ServerCall* qui défini la méthode *OnFailure()*, nous n'avons donc plus qu'à redéfinir la méthode *OnSuccess()* qui elle effectue un traitement différent à chaque fois.

5.3.8 Génération du rapport

Le logiciel permet de générer le rapport en format HTML et PDF. Le format HTML est plus accessible dans le sens qu'il donne à l'utilisateur une possibilité de faire de nouvelles modifications, en fournissant des changements dans le code HTML si nécessaire. L'implémentation de cette fonctionnalité est fournie par le *ReportServiceImpl*, sur le côté serveur. Le service crée un fichier HTML avec des styles CSS. Par la suite, il remplit le fichier avec les données choisies par l'utilisateur. Le résultat est affiché immédiatement dans une nouvelle fenêtre du navigateur internet.

Le format PDF peut être le choix de l'utilisateur quand il ya un besoin de présenter ou de conserver les informations générées. Pour l'implémentation de cette fonctionnalité, nous avons utilisé une bibliothèque Java pour le rendu XML, XHTML et CSS 2.1 : Flying Saucer

"Flying Saucer" est souvent utilisé comme une bibliothèque côté serveur pour générer des documents PDF, car il est capable de convertir un fichier XHTML au format PDF. De cette façon, nous pouvons utiliser le fichier HTML créé par le *ReportServiceImpl* pour générer le document PDF. Nous avons créé le sevlet *PDFCreator* qui parse le fichier HTML dans un document DOM à l'aide de l'api Java XML. Par la suite le servlet définit le document DOM en tant que document de l'objet *ITextRenderer*. L'objet *ITextRenderer* est défini par une bibliothèque qui vient avec Flying Saucer. Cet objet est

donc responsable de la création du document PDF. La dernière chose à faire est d'envoyer le document PDF créé à l'objet *OutputStream* du servlet, afin que le document puisse être affiché immédiatement dans une nouvelle fenêtre du navigateur internet.

Chapitre 6

Tests

6.1 Tests à scénarios

Le scénario suivant a été réalisé sur différents navigateurs et système d'exploitations afin de prouver la portabilité de notre logiciel.

1. Zoomer avec la molette de la souris sur le site qui vous intéresse.
2. Ajouter des points sur la carte en appuyant sur le bouton correspondant.
3. Éditer quelques points.
4. Ajouter des points GPS via le bouton add de la liste des points.
5. Créer et éditer des lignes de la même manière.
6. Créer des zones contenant des lignes et/ou d'autres zones.
7. Ajouter des unités de mesure.
8. Sauvegarder le projet.
9. Supprimer des points et des lignes via la carte et les listes.
10. Supprimer des zones via la liste.
11. Créer un nouveau projet.
12. Charger le projet préalablement sauvegardé.
13. Sélectionner plusieurs zones et y effectuer le calcul.
14. Générer un rapport avec différents paramètres en HTML et en PDF.
15. Exporter le projet en KML.
16. Ouvrir le fichier KML avec GoogleEarth.
17. Importer un fichier KML.
18. Tester toutes les langues disponibles.

6.2 Tests Unitaires

6.2.1 XMLServiceTest et KMLServicesTest

Les tests réalisés sur les services qui gèrent l'importation et l'exportation de données (au format XML et KML) permettent de valider les besoins fonctionnels associés.

Tous les tests réalisés commencent par créer un ensemble de donnée, de les sauvegarder dans un fichier (KML et XML) puis nous importons ces données, que nous comparons ensuite aux données d'entrée.

Nous avons effectuer des tests aux limites, en testant :

- des ensembles de données vide
- des ensembles de données de taille supérieure à la limite de 1000 entités géographique.

Les fichiers XML sont vérifiés conforme à la norme XML 1.0 par le vérificateur du W3C (http ://www.w3.org/2001/03/webdata/xsv).

Les fichiers KML sont eux vérifiés conforme à la norme KML 2.2 défini par le OGC (http ://www.kmlvalidator.com/home.htm).

6.2.2 EntitiesServiceTest

Cet ensemble de test permet de tester le fonctionnement du service de gestion des entités géographique.

Nous avons fait très attention à réaliser de nombreux test de domaine et notamment des tests aux limites.

Les tests OutOfBoundsValues* permettent de vérifier que les paramètres de latitude et longitude des méthodes gérant les points GPS renvoient bien des exceptions (OvertakenBoundsException) en cas de valeurs qui sortent de leur domaine de définition ([-90 ;90] pour la latitude et [-180 ;180] pour la longitude).

6.3 Tests de Performance

Pour enregistrer les performances d'une application codée avec GWT, Google a developpé un plugin pour son navigateur Chrome, appelé Speed Tracer.

Grâce à ce plugin, il est possible de quantifier la fluidité de l'interface graphique, et de visualiser les appels au serveur, et l'engorgement de ces appels (ce qui nous a permis par la suite d'optimiser certains points de notre programme).

Notre application n'étant pas déployée sur un serveur pour l'instant, ces tests de performances ne prennent pas en compte le temps de transmission des données entre le client et le serveur.

6.3.1 Conditions d'utilisation normales

1. **Ajout d'un point**

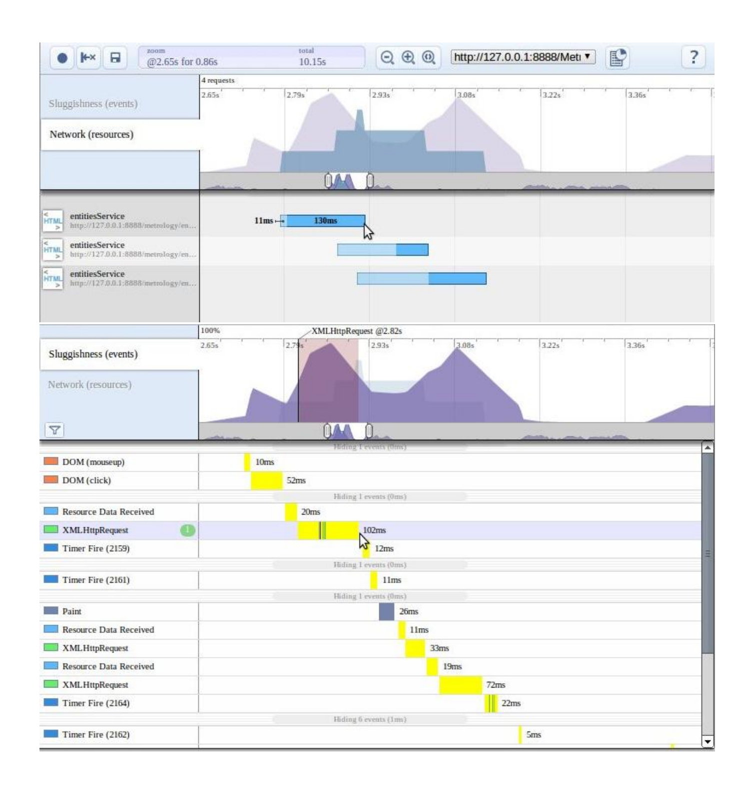


FIG. 6.1 – Speed Tracer

Le premier graphique de la figure 6.1 nous permet de visualiser les appels à l'entitiesService. Lors de l'ajout d'un point, on constate que trois appels sont réalisés, le premier pour ajouter le point sur le serveur, et les deux autres pour actualiser respectivement la carte et la liste des points.

Le second graphique permet de voir la suite d'évènements qui se produisent coté client, ainsi que le ralentissement dû à l'interface graphique. Les évènements "XMLHttpRequest", précédés d'un "Resource Data Received", correspondent à la réception de la réponse du serveur par le client.

On constate donc que dans des conditions normales, l'ajout d'un point via le menu prend 380ms, et ceci sans prendre en compte le temps de latence entre le client et le serveur, qui est supposé nul ici, étant donné que les deux sont sur la même machine.

La rapidité n'étant pas une contrainte de notre application, nous nous satisferons de ce temps d'exécution. Mais on peut tout de même se demander s'il ne serait pas possible de réduire ce temps d'exécution, par exemple en ne faisant qu'un seul appel au serveur, qui renvoie la liste des points (liste qui serait alors utilisée par le MapPresenter et le GPSPointsPresenter pour actualiser leurs vues respectives), au lieu de laisser chaque présenteur faire ses appels au serveur.

6.3.2 Conditions de stress

1. **Ajout de cent points**
   Pour réaliser ce test, on ajoute un boucle for dans le EditGPSPoints-Presenter pour qu'il ajoute cent fois le même point quand on clique sur le bouton "Save".
   En faisant différents tests, avec ou sans l'actualisation de la carte, on se rend compte que l'actualisation de la carte prend beaucoup de temps, comparé à l'actualisation de la liste des points
2. **Ajout de cent lignes**
   La méthode utilisée pour ajouter une ligne étant similaire à l'ajout d'un point, les résultats des tests sont plus ou moins les mêmes, et on constate ici aussi que l'actualisation de la carte est plus lente que le reste.
3. **Ajout de cent unités de mesure**
   L'ajout de cent unités de mesure fonctionne et prend environ 150 secondes.
4. **Import d'un fichier KML**
   Nous avons également testé l'import d'un fichier KML contenant un grand nombre de lignes (plus de 1000), et il s'avère encore une fois que la carte est la partie la plus lente de notre application, et qu'elle gagnerait à être optimisée.

6.3.3 Conclusion des tests

Ces tests de performance nous ont permis de tester la partie client de notre application, ce qui n'a pas été fait avec les tests unitaire, qui ne testaient que

la partie serveur, et également d'optimiser certains points, et de voir quels points peuvent être optimisés dans le futur.

Chapitre 7

Bilan

7.1 Planning effectif

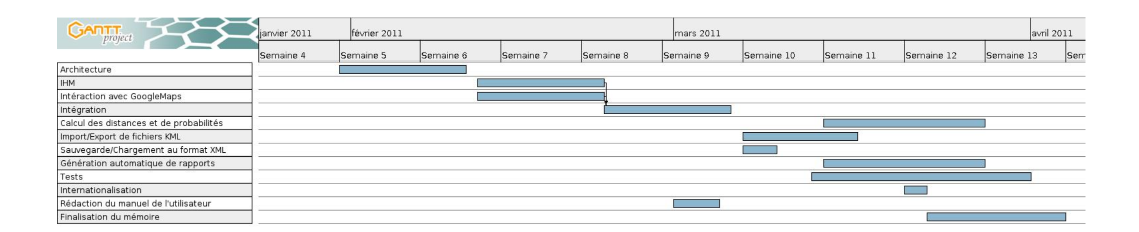


FIG. 7.1 – Planning effectif

Le planning prévisionnel a été respecté les premières semaines, avec même un peu d'avance concernant l'intégration de GoogleMaps, cependant, cette avance à été vite comblée par le temps perdu à programmer certaines vues complexes, notamment celle qui affiche les zones.
De la même manière, la sauvegarde et chargement des fichiers XML à pris très peu de temps, mais l'import et l'export des fichiers KML a pris plus de temps à être développé, étant donné que la quantité et la variété d'informations pouvant être contenue dans ce type de fichiers est élevée.
Globalement, le planning à été respecté : nous avons fini dans les temps, même si certaines parties de l'application peuvent être améliorées dans le futur.

7.2 Expérience personnelle

7.3 Perspectives

7.3.1 Futur de l'application

Dû aux contraintes de temps, l'application, bien que fonctionnelle, n'est toujours pas en ligne. Nous n'avons donc pas encore de retour du client, et il se peut que des modifications soit demandées. pour le même raison, les tests d'ergonomie n'ont pas encore été réalisés.
Nous avons vu dans la section 6.3 que l'import des gros fichiers KML était grandement ralenti par la mise à jour de la carte, c'est donc un point qui pourra être amélioré dans le futur, par exemple en ajoutant une option à la carte, qui permettrait de choisir d'afficher ou non les points GPS, ce qui réduirait le nombre d'entités à charger en même temps sur la carte.

7.3.2 Et si on recommençait ?

Un choix que nous avons fait et dont nous sommes satisfaits, est l'élection de GWT comme api pour programmer cette application. En revanche, les widgets inclus dans GWT sont assez sommaires, et nous avons passé beaucoup de temps à développer les vues, c'est pourquoi l'utilisation d'une bibliothèque tierce définissant des widgets plus avancés, comme Ext GWT[1] par exemple nous permettrait de gagner beaucoup de temps.

[1] http ://www.sencha.com/products/extgwt/

Annexe A

Le code des Tests

A.1 Analyse Service Test

```
@Test
 public void thousandPointsMeasuresTest() {
   try {
     List<Entity> points = new ArrayList<Entity>();
     List<Measure> measures = new ArrayList<Measure>();

     Measure meter = new Measure("Meter", "m", 1f, "m");
     measures.add(meter);

     Random random = new Random();

     //:COMMENT:Arnaud:21/03/2011: Created measures
     for (int i = 0; i < 1000; i++) {
       Measure m = new Measure("Measure"+i, "smb"+i, random.nextFloat()
       measures.add(m);
     }

     //:COMMENT:Arnaud:21/03/2011: Created points
     for (int i = 0; i < 1000; i++) {
       GPSPoint p = new GPSPoint("point" + Integer.toString(i), random
           .nextDouble(), random.nextDouble());
       p.setId(i);
       points.add(p);
     }

     //:COMMENT:Arnaud:23/03/2011: Create XML
```

```
    XMLServiceImpl xmlService = new XMLServiceImpl();
    String path = xmlService.exportXML("testExportXML1.xml", points, m

    //:COMMENT:Arnaud:22/03/2011: Load entities from XML
    List<Entity> points2 =
      xmlService.importXML(path);
    List<Measure> measures2 = xmlService.getMeasures();

    if ((points.size() != points2.size()) ||
        (measures.size() != measures2.size())) {
      fail("measures␣:"+measures.size() +"␣and␣"+
          measures2.size());
    }


  } catch (XMLExportException e) {
    fail("XMLExportException␣!");
  } catch (NullArgumentException e) {
    fail("NullArgumentException");
  } catch (XMLImportException e) {
    fail("XMLImportException");
  } catch (AlreadyUsedIdException e) {
    fail("AlreadyUsedIdException");
  }

}
```

Annexe B

Questionnaire

Goal

The main goal of this questionnary is to give us a feedback of our software to permit to improve it. The best way to help us is to do a simple work with our sotware and then fill a short questionnaire about the thing which have to be improved or changed.

Work to do

Your first job is to read the user guide, you shouldn't spend more than thirty minutes to read it. Then you have to import a given KML file in the software which contains 8 points that represents the corner of two fields in Ruweiha (a archeologic site in Syria). You have to create lines in the border of these fields and to embeded them into 2 areas. Next you have to create a "super" area that contains to two other. Now just select the "super" area and compute the possible measures used. After that generate a pdf report.

Questions

Please answer the next quetions to help us to improve our software :
How much time have you spent on learning how to use the software ?
....
Do you find the user guide is helpfull ?
....
How much time have you spent to do the work above ?
....

What do you think about the graphical interface ?

....

Is the software easy to use ?

....

Is the information given in the graphical interface enough ?

....

If no, what do you suggest to add ?

....

What do you think about the design of the report ?

....

Is the information complete in the report ?

....

How do you find the map ?

....

Comments :

....

....

....

....